Amazon FreeRTOS

User Guide
# Table of Contents

What Is Amazon FreeRTOS? ................................................................................................................................. 1  
The FreeRTOS Kernel ............................................................................................................................................. 1  
Amazon FreeRTOS Libraries ................................................................................................................................... 1  
Amazon FreeRTOS Console ....................................................................................................................................... 1  
Downloading Amazon FreeRTOS Source Code ...................................................................................................... 2  
Over-the-Air Updates ............................................................................................................................................... 2  
Development Workflow ....................................................................................................................................... 2  
Getting Started with Amazon FreeRTOS .................................................................................................................. 4  
Prerequisites ............................................................................................................................................................ 4  
  AWS Account and Permissions ...................................................................................................................................... 4  
  Amazon FreeRTOS Supported Hardware Platforms .................................................................................................. 5  
  Registering Your MCU Board with AWS IoT ............................................................................................................ 5  
  Install a Terminal Emulator ...................................................................................................................................... 7  
Getting Started with the Texas Instruments CC3220SF-LAUNCHXL ........................................................................... 7  
  Setting Up Your Environment .................................................................................................................................. 7  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 9  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 11  
  Troubleshooting ....................................................................................................................................................... 12  
Getting Started with the STMicroelectronics STM32L4 Discovery Kit IoT Node ................................................. 13  
  Setting Up Your Environment .................................................................................................................................. 13  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 13  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 15  
  Troubleshooting ....................................................................................................................................................... 16  
Getting Started with the NXP LPC54018 IoT Module ................................................................................................. 16  
  Setting Up Your Environment .................................................................................................................................. 16  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 17  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 19  
  Troubleshooting ....................................................................................................................................................... 20  
Getting Started with the Microchip Curiosity PIC32MZEF ......................................................................................... 20  
  Setting Up the Microchip Curiosity PIC32MZEF Hardware ............................................................................................ 20  
  Setting Up Your Environment .................................................................................................................................. 21  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 22  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 23  
  Troubleshooting ....................................................................................................................................................... 25  
Getting Started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT ....................................................... 25  
  Setting Up the Espressif Hardware ............................................................................................................................ 25  
  Setting Up Your Environment .................................................................................................................................. 26  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 26  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 27  
  Troubleshooting ....................................................................................................................................................... 31  
Getting Started with the Infineon XMC4800 IoT Connectivity Kit ............................................................................ 36  
  Setting Up Your Environment .................................................................................................................................. 36  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 37  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 38  
Getting Started with the Xilinx Avnet MicroZed Industrial IoT Kit ............................................................................ 40  
  Setting Up the MicroZed Hardware ............................................................................................................................ 41  
  Setting Up Your Environment .................................................................................................................................. 41  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 43  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 44  
  Troubleshooting ....................................................................................................................................................... 52  
Getting Started with the FreeRTOS Windows Simulator ........................................................................................... 52  
  Setting Up Your Environment .................................................................................................................................. 52  
  Download and Configure Amazon FreeRTOS ........................................................................................................... 52  
  Build and Run Amazon FreeRTOS Samples ........................................................................................................... 54
# Amazon FreeRTOS User Guide

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting Started with the Nordic nRF52840-DK</td>
<td>55</td>
</tr>
<tr>
<td>Setting Up the Nordic Hardware</td>
<td>55</td>
</tr>
<tr>
<td>Setting Up Your Environment</td>
<td>55</td>
</tr>
<tr>
<td>Download and Configure Amazon FreeRTOS</td>
<td>55</td>
</tr>
<tr>
<td>Build and Run Amazon FreeRTOS Samples</td>
<td>57</td>
</tr>
<tr>
<td>Amazon FreeRTOS Developer Guide</td>
<td>58</td>
</tr>
<tr>
<td>Amazon FreeRTOS Architecture</td>
<td>58</td>
</tr>
<tr>
<td>FreeRTOS Kernel Fundamentals</td>
<td>58</td>
</tr>
<tr>
<td>FreeRTOS Kernel Scheduler</td>
<td>59</td>
</tr>
<tr>
<td>Memory Management</td>
<td>59</td>
</tr>
<tr>
<td>Intertask Coordination</td>
<td>60</td>
</tr>
<tr>
<td>Software Timers</td>
<td>63</td>
</tr>
<tr>
<td>Low Power Support</td>
<td>63</td>
</tr>
<tr>
<td>Amazon FreeRTOS Libraries</td>
<td>64</td>
</tr>
<tr>
<td>Amazon FreeRTOS Porting Libraries</td>
<td>64</td>
</tr>
<tr>
<td>Amazon FreeRTOS Application Libraries</td>
<td>69</td>
</tr>
<tr>
<td>Bluetooth Low Energy</td>
<td>72</td>
</tr>
<tr>
<td>AWS IoT Device Defender</td>
<td>82</td>
</tr>
<tr>
<td>AWS IoT Greengrass</td>
<td>85</td>
</tr>
<tr>
<td>MQTT (Beta)</td>
<td>87</td>
</tr>
<tr>
<td>MQTT (Legacy)</td>
<td>90</td>
</tr>
<tr>
<td>Over-the-Air (OTA) Agent</td>
<td>93</td>
</tr>
<tr>
<td>Public Key Cryptography Standard (PKCS) #11</td>
<td>95</td>
</tr>
<tr>
<td>Secure Sockets</td>
<td>97</td>
</tr>
<tr>
<td>AWS IoT Device Shadow</td>
<td>101</td>
</tr>
<tr>
<td>Transport Layer Security (TLS)</td>
<td>103</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>103</td>
</tr>
<tr>
<td>Amazon FreeRTOS Over-the-Air Updates</td>
<td>108</td>
</tr>
<tr>
<td>Over-the-Air Update Prerequisites</td>
<td>108</td>
</tr>
<tr>
<td>OTA Tutorial</td>
<td>123</td>
</tr>
<tr>
<td>OTA Update Manager Service</td>
<td>143</td>
</tr>
<tr>
<td>Integrating the OTA Agent into Your Application</td>
<td>144</td>
</tr>
<tr>
<td>OTA Security</td>
<td>147</td>
</tr>
<tr>
<td>OTA Troubleshooting</td>
<td>147</td>
</tr>
<tr>
<td>Amazon FreeRTOS Console User Guide</td>
<td>153</td>
</tr>
<tr>
<td>Managing Amazon FreeRTOS Configurations</td>
<td>153</td>
</tr>
<tr>
<td>Amazon FreeRTOS Demo Projects</td>
<td>156</td>
</tr>
<tr>
<td>Navigating the Demo Applications</td>
<td>156</td>
</tr>
<tr>
<td>Directory and File Organization</td>
<td>156</td>
</tr>
<tr>
<td>Configuration Files</td>
<td>157</td>
</tr>
<tr>
<td>Bluetooth Low Energy Demo Applications (Beta)</td>
<td>157</td>
</tr>
<tr>
<td>Overview</td>
<td>158</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>158</td>
</tr>
<tr>
<td>Common Components</td>
<td>159</td>
</tr>
<tr>
<td>MQTT over BLE</td>
<td>160</td>
</tr>
<tr>
<td>Wi-Fi Provisioning</td>
<td>162</td>
</tr>
<tr>
<td>Generic Attributes Server</td>
<td>164</td>
</tr>
<tr>
<td>Secure Sockets Echo Client Demo</td>
<td>165</td>
</tr>
<tr>
<td>Device Shadow Demo Application</td>
<td>166</td>
</tr>
<tr>
<td>Greengrass Discovery Demo Application</td>
<td>167</td>
</tr>
<tr>
<td>OTA Demo Application</td>
<td>168</td>
</tr>
<tr>
<td>Demo Bootloader for the Microchip Curiosity PIC32MZEF</td>
<td>171</td>
</tr>
<tr>
<td>Bootloader States</td>
<td>171</td>
</tr>
<tr>
<td>Flash Device</td>
<td>172</td>
</tr>
<tr>
<td>Application Image Structure</td>
<td>173</td>
</tr>
<tr>
<td>Image Header</td>
<td>173</td>
</tr>
<tr>
<td>Image Descriptor</td>
<td>174</td>
</tr>
</tbody>
</table>
What Is Amazon FreeRTOS?

Amazon FreeRTOS consists of the following components:

- A microcontroller operating system based on the FreeRTOS kernel
- Amazon FreeRTOS libraries for connectivity, security, and over-the-air (OTA) updates.
- A console that allows you to download a zip file that contains everything you need to get started with Amazon FreeRTOS.
- Over-the-air (OTA) Updates.
- The Amazon FreeRTOS Qualification Program.

The FreeRTOS Kernel

The FreeRTOS kernel is a real-time operating system kernel that supports numerous architectures and is ideal for building embedded microcontroller applications. The kernel provides:

- A multitasking scheduler.
- Multiple memory allocation options (including the ability to create statically allocated systems).
- Inter-task coordination primitives, including task notifications, message queues, multiple types of semaphores, and stream and message buffers.

Amazon FreeRTOS Libraries

Amazon FreeRTOS includes libraries that enable you to:

- Securely connect devices to the AWS IoT cloud using MQTT and device shadows.
- Discover and connect to AWS IoT Greengrass cores.
- Manage Wi-Fi connections.
- Audit the configuration of your devices, monitor connected devices to detect abnormal behavior, and to mitigate security risks. For more information, see AWS IoT Device Defender. Amazon FreeRTOS provides a library that enables your Amazon FreeRTOS-based devices to write metrics to AWS IoT Device Defender. For more information, see Amazon FreeRTOS Device Defender Library.

Note
The Device Defender library currently works on the Microchip Curiosity PIC32MZEF development board and the Windows simulator.

- Listen for and process over-the-air (OTA) updates.

Amazon FreeRTOS Console

The Amazon FreeRTOS console enables you to configure and download a package that contains everything you need to write an application for your microcontroller-based devices:

- The FreeRTOS kernel
- Amazon FreeRTOS libraries
- Platform support libraries
• Hardware drivers

You can download a package with a predefined configuration or create your own configuration by selecting your hardware platform and the libraries required for your application. These configurations are saved in AWS and are available for download at any time.

The Amazon FreeRTOS console is part of the AWS IoT console. You can find it by choosing the link above or by browsing to the AWS IoT console.

To open the Amazon FreeRTOS console
1. Browse to the AWS IoT console.
2. From the navigation pane choose Software.

Downloading Amazon FreeRTOS Source Code

You can download the RTOS kernel and software libraries from the Amazon FreeRTOS console or from GitHub.

Over-the-Air Updates

Internet-connected devices can be in use for a long time, and must be updated periodically to fix bugs and improve functionality. Often these devices must be updated in the field and need to be updated remotely or "over-the-air". The Amazon FreeRTOS Over-the-Air (OTA) Update service enables you to:

• Digitally sign firmware prior to deployment.
• Securely deploy new firmware images to a single device, a group of devices, or your entire fleet.
• Deploy firmware to devices as they are added to groups, reset, or reprovisioned.
• Once deployed to devices, verify the authenticity and integrity of the new firmware.
• Monitor the progress of a deployment.
• Debug a failed deployment.

When you send files over the air, it is a best practice to digitally sign them so that the devices that receive the files can verify they have not been tampered with en route. You can use Code Signing for Amazon FreeRTOS to sign and encrypt your files or you can sign your files with your own code-signing tools. For more information about Code Signing for Amazon FreeRTOS, see the Code Signing for Amazon FreeRTOS Developer Guide.

For more information about OTA updates, see:
• Amazon FreeRTOS Over-the-Air Updates (p. 108)
• OTA Demo Application (p. 168)

Development Workflow

You start development by configuring and downloading a package from the Amazon FreeRTOS console in the AWS IoT console. You unzip the package and import it into your IDE. You can then develop your embedded application on your selected hardware platform and manufacture and deploy these devices
using the development process appropriate for your device. Deployed devices can connect to the AWS IoT service or AWS IoT Greengrass as part of a complete IoT solution. The following diagram shows the development workflow and the subsequent connectivity from Amazon FreeRTOS-based devices.

You can also download the Amazon FreeRTOS source code from GitHub.
Getting Started with Amazon FreeRTOS

This section shows you how to download and configure Amazon FreeRTOS and run a demo application on one of the qualified microcontroller boards. In this tutorial, we assume you are familiar with AWS IoT and the AWS IoT console. If not, we recommend that you start with the AWS IoT Getting Started tutorial.

Prerequisites

To follow along with this tutorial, you need an AWS account, an IAM user with permission to access AWS IoT and Amazon FreeRTOS, and one of the supported hardware platforms.

AWS Account and Permissions

To create an AWS account, see Create and Activate an AWS Account.

To add an IAM user to your AWS account, see IAM User Guide. To grant your IAM user account access to AWS IoT and Amazon FreeRTOS, attach the following IAM policies to your IAM user account:

- AmazonFreeRTOSFullAccess
- AWSIoTFullAccess

To attach the AmazonFreeRTOSFullAccess policy to your IAM user

1. Browse to the IAM console, and from the navigation pane, choose Users.
2. Enter your user name in the search text box, and then choose it from the list.
3. Choose Add permissions.
4. Choose Attach existing policies directly.
5. In the search box, enter AmazonFreeRTOSFullAccess, choose it from the list, and then choose Next: Review.
6. Choose Add permissions.

To attach the AWSIoTFullAccess policy to your IAM user

1. Browse to the IAM console, and from the navigation pane, choose Users.
2. Enter your user name in the search text box, and then choose it from the list.
3. Choose Add permissions.
4. Choose Attach existing policies directly.
5. In the search box, enter AWSIoTFullAccess, choose it from the list, and then choose Next: Review.
6. Choose Add permissions.

For more information about IAM and user accounts, see IAM User Guide.
For more information about policies, see IAM Permissions and Policies.

Amazon FreeRTOS Supported Hardware Platforms

You need one of the supported MCU boards:

- STMicroelectronics STM32L4 Discovery Kit IoT Node
- Texas Instruments CC3220SF-LAUNCHXL
- NXP LPC54018 IoT Module
- Microchip Curiosity PIC32MZEF Bundle
- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- Infineon XMC4800 IoT Connectivity Kit
- Xilinx Avnet MicroZed Industrial IoT Kit
- Microsoft Windows 7 or later, with at least a dual core and a hard-wired Ethernet connection
- Nordic nRF52840-DK [BETA]

Registering Your MCU Board with AWS IoT

You must register your MCU board so it can communicate with AWS IoT. To register your device, you must create:

- An IoT thing.
  
  An IoT thing allows you to manage your devices in AWS IoT.
- A private key and X.509 certificate.
  
  The private key and certificate allow your device to authenticate with AWS IoT.
- An AWS IoT policy.
  
  The AWS IoT policy grants your device permissions to access AWS IoT resources.

To create an AWS IoT policy

1. To create an IAM policy, you need to know your AWS Region and AWS account number.

   To find your AWS account number, in the upper-right corner of the AWS Management Console, choose My Account. Your account ID is displayed under Account Settings.

   To find the AWS Region for your AWS account, open a command prompt window and enter the following command:

   ```
   AWS IoT describe-endpoint
   ```

   The output should look like this:

   ```
   {
     "endpointAddress": "xxxxxxxxxx.x.us-west-2.amazonaws.com"
   }
   ```

   In this example, the region is `us-west-2`. 
2. Browse to the **AWS IoT** console.
3. In the navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**.
4. Enter a name to identify your policy.
5. In the **Add statements** section, choose **Advanced mode**. Copy and paste the following JSON into the policy editor window. Replace `aws-region` and `aws-account` with your region and account ID.

   ```json
   {
   "Version": "2012-10-17",
   "Statement": [
   {
   "Effect": "Allow",
   "Action": "iot:Connect",
   },
   {
   "Effect": "Allow",
   "Action": "iot:Publish",
   },
   {
   "Effect": "Allow",
   "Action": "iot:Subscribe",
   },
   {
   "Effect": "Allow",
   "Action": "iot:Receive",
   }
   ]
   }
   ```

   This policy grants the following permissions:

   - **iot:Connect**
     Grants your device the permission to connect to the AWS IoT message broker.
   - **iot:Publish**
     Grants your device the permission to publish an MQTT message on the `freertos/demos/echo` MQTT topic.
   - **iot:Subscribe**
     Grants your device the permission to subscribe to the `freertos/demos/echo` MQTT topic filter.
   - **iot:Receive**
     Grants your device the permission to receive messages from the AWS IoT message broker.

6. Choose **Create**.

---

**To create an IoT thing, private key, and certificate for your device**

1. Browse to the **AWS IoT** console.
2. In the navigation pane, choose **Manage**, and then choose **Things**.
3. If you do not have any IoT things registered in your account, the **You don't have any things yet** page is displayed. If you see this page, choose **Register a thing**. Otherwise, choose **Create**.
4. On the **Creating AWS IoT things** page, choose **Create a single thing**.
Install a Terminal Emulator

A terminal emulator can help you diagnose problems or verify your device code is running properly. There are a variety of terminal emulators available for Windows, macOS, and Linux. You must connect your device to your computer before you attempt to connect a terminal emulator to your device.

Use these settings in your terminal emulator:

<table>
<thead>
<tr>
<th>Terminal Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>Depends on platform and other devices you have connected to your computer.</td>
</tr>
<tr>
<td>BAUD rate</td>
<td>115200</td>
</tr>
<tr>
<td>Data</td>
<td>8 bit</td>
</tr>
<tr>
<td>Parity</td>
<td>none</td>
</tr>
<tr>
<td>Stop</td>
<td>1 bit</td>
</tr>
<tr>
<td>Flow control</td>
<td>none</td>
</tr>
</tbody>
</table>

**Getting Started with the Texas Instruments CC3220SF-LAUNCHXL**

Before you begin, see Prerequisites (p. 4).

If you do not have the Texas Instruments (TI) CC3220SF-LAUNCHXL Development Kit, you can purchase one from Texas Instruments.

**Setting Up Your Environment**

Amazon FreeRTOS supports two IDEs for the TI CC3220SF-LAUNCHXL development kit: Code Composer Studio and IAR Embedded Workbench.

For information about installing Code Composer Studio, see Install Code Composer Studio (p. 8).

For information about installing IAR Embedded Workbench, see Install IAR Embedded Workbench (p. 8).
You also need to **Install the SimpleLink CC3220 SDK (p. 8), Install Uniflash (p. 9), Configure Wi-Fi Provisioning (p. 9), and Install the Latest Service Pack (p. 9).**

### Install Code Composer Studio

1. Browse to **TI Code Composer Studio**.
2. Download the offline installer for version 7.3.0 for the platform of your host machine (Windows, macOS, or Linux 64-bit).
3. Unzip and run the offline installer. Follow the prompts.
4. For **Product Families to Install**, choose **SimpleLink Wi-Fi CC32xx Wireless MCUs**.
5. On the next page, accept the default settings for debugging probes, and then choose **Finish**.

If you experience issues when you are installing Code Composer Studio, see **TI Development Tools Support, Code Composer Studio FAQs**, and **Troubleshooting Code Composer Studio**.

### Install IAR Embedded Workbench

1. Browse to **IAR Embedded Workbench for ARM**.
2. Download and run the Windows installer. In **Debug probe drivers**, make sure that **TI XDS** is selected:

![IAR Embedded Workbench for ARM 8.22.1](image)

3. Complete the installation and launch the program. On the **License Wizard** page, choose **Register with IAR Systems to get an evaluation license**, or use your own IAR license.

### Install the SimpleLink CC3220 SDK

Install the **SimpleLink CC3200 SDK**. The SimpleLink Wi-Fi CC3200 SDK contains drivers for the CC3200 programmable MCU, more than 40 sample applications, and documentation required to use the samples.
Install Uniflash

Install Uniflash. CCS Uniflash is a standalone tool used to program on-chip flash memory on TI MCUs. Uniflash has a GUI, command line, and scripting interface.

Configure Wi-Fi Provisioning

To configure the Wi-Fi settings for your board, do one of the following:

- Complete the Amazon FreeRTOS demo application described in Configure Your Project (p. 10).
- Use SmartConfig from Texas Instruments.

Install the Latest Service Pack

1. On your TI CC3220SF-LAUNCHXL, place the SOP jumper on the middle set of pins (position = 1) and reset the board.
2. Start Uniflash, and from the list of configurations, choose CC3200SF-LAUNCHXL. Choose Start Image Creator.
3. Choose New Project.
4. On the Start new project page, enter a name for your project. For Device Type, choose CC3220SF. For Device Mode, choose Develop, and then choose Create Project.
5. Disconnect your serial terminal (if previously connected) and on the right side of the Uniflash application window, choose Connect.
6. From the left column, choose Service Pack.
7. Choose Browse, and then navigate to where you installed the CC3220SF SimpleLink SDK. The service pack is located at ti\simplelink_cc32xx_sdk\tools\cc32xx_tools\servicepack-cc3x20\sp_VERSION.bin.
8. Choose the button and then choose Program Image (Create & Program) to install the service pack. Remember to switch the SOP jumper back to position 0 and reset the board.

Download and Configure Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS.

Download Amazon FreeRTOS

1. Browse to the AWS IoT console.
2. In the navigation pane, choose Software.
4. Under Software Configurations, find Connect to AWS IoT- TI, and then:
   - If you are using Code Composer Studio, choose Download.
   - If you are using IAR Embedded Workbench, choose Connect to AWS IoT-TI. Under Hardware platform, choose Edit. Under Integrated Development Environment (IDE), choose IAR Embedded Workbench. Make sure the compiler is set to IAR, and then choose Create and Download.
5. Unzip the downloaded file to your hard drive. When unzipped, you have a directory named AmazonFreeRTOS. You can place this directory anywhere you want, but be aware of path length limitations on Windows.
Note
The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, C:\Users\Username\Dev\AmazonFreeRTOS works, but C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS causes build failures.
In this tutorial, the path to the AmazonFreeRTOS directory is referred to as BASE_FOLDER.

Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. Registering Your MCU Board with AWS IoT (p. 5) is a step in the Prerequisites (p. 4).

To configure your AWS IoT endpoint
1. Browse to the AWS IoT console.
2. In the navigation pane, choose Settings.

   Your AWS IoT endpoint is displayed in Endpoint. It should look like <1234567890123>-ats.iot.<us-east-1>.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose Manage, and then choose Things.

   Your device should have an AWS IoT thing name. Make a note of this name.
4. In your IDE, open <BASE_FOLDER>\demos\common\include\aws_clientcredential.h and specify values for the following #define constants:
   - clientcredentialMQTT_BROKER_ENDPOINT Your AWS IoT endpoint
   - clientcredentialIOT_THING_NAME The AWS IoT thing name of your board

To configure your Wi-Fi
1. Open the aws_clientcredential.h file.
2. Specify values for the following #define constants:
   - clientcredentialWIFI_SSID The SSID for your Wi-Fi network
   - clientcredentialWIFI_PASSWORD The password for your Wi-Fi network
   - clientcredentialWIFI_SECURITY The security type of your Wi-Fi network

   Valid security types are:
   - eWiFiSecurityOpen (Open, no security)
   - eWiFiSecurityWEP (WEP security)
   - eWiFiSecurityWPA (WPA security)
   - eWiFiSecurityWPA2 (WPA2 security)

To configure your AWS IoT credentials

Note
To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.
Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You must format the certificate and private key for your device.

1. In a browser window, open `<BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the `<ID>-certificate.pem.crt` that you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the `<ID>-private.pem.key` that you downloaded from the AWS IoT console.
4. Choose **Generate and save aws_clientcredential_keys.h**, and then save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the existing file in the directory.

*Note*

The certificate and private key are hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.

### Build and Run Amazon FreeRTOS Samples

#### Build and Run Amazon FreeRTOS Samples in TI Code Composer

**Import the Amazon FreeRTOS Sample Code into TI Code Composer**

1. Open TI Code Composer, and choose **OK** to accept the default workspace name.
2. On the **Getting Started** page, choose **Import Project**.
3. In **Select search-directory**, enter `<BASE_FOLDER>\demos\ti\cc3220_launchpad\ccs`. The project `aws_demos` should be selected by default. To import the project into TI Code Composer, choose **Finish**.
4. In **Project Explorer**, double-click `aws_demos` to make the project active.
5. From **Project**, choose **Build Project** to make sure the project compiles without errors or warnings.

**Subscribe to MQTT topic**

*Note*

Before you run the Amazon FreeRTOS samples, do the following:

1. Make sure the Sense On Power (SOP) jumper on your Texas Instruments CC3220SF-LAUNCHXL is in position 0. For more information, see **CC3200 SimpleLink User's Guide**.
2. Use a USB cable to connect your Texas Instruments CC3220SF-LAUNCHXL to your computer.
3. Sign in to the **AWS IoT console**.
4. In the navigation pane, choose **Test** to open the MQTT client.
5. In **Subscription topic**, enter `freertos/demos/echo`, and then choose **Subscribe to topic**.

**Run the Amazon FreeRTOS samples in TI Code Composer**

1. Rebuild your project.
2. In TI Code Composer, from **Run**, choose **Debug**.
3. When the debugger stops at the breakpoint in `main()`, go to the **Run** menu, and choose **Resume**.

In the MQTT client in the AWS IoT console, you should see the MQTT messages sent by your device.
Troubleshooting

If you don’t see messages in the MQTT client of the AWS IoT console, you might need to configure debug settings for the board.

1. In Code Composer, on Project Explorer, choose aws_demos.
2. From the Run menu, choose Debug Configurations.
3. In the navigation pane, choose aws_demos.
4. On the Target tab, under Connection Options, choose Reset the target on a connect.
5. Choose Apply, and then choose Close.

If these steps don’t work, look at the program’s output in the serial terminal. You should see some text that indicates the source of the problem.
Getting Started with the STMicroelectronics STM32L4 Discovery Kit IoT Node

Before you begin, see Prerequisites (p. 4).

If you do not already have the STMicroelectronics STM32L4 Discovery Kit IoT Node, you can purchase one from STMicroelectronics.

Make sure you have installed the latest Wi-Fi firmware. To download the latest Wi-Fi firmware, see STM32L4 Discovery kit IoT node, low-power wireless, BLE, NFC, SubGHz, Wi-Fi. Under Binary Resources, choose Inventek ISM 43362 Wi-Fi module firmware update (read the readme file for instructions).

Setting Up Your Environment

Install System Workbench for STM32

1. Browse to OpenSTM32.org.
2. Register on the OpenSTM32 webpage. You need to sign in to download System Workbench.

If you experience issues during installation, see the FAQs on the System Workbench website.

Download and Configure Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS.

Download Amazon FreeRTOS

1. In the AWS IoT console, browse to the Amazon FreeRTOS page.
2. In the navigation pane, choose Software.
4. Choose Download FreeRTOS Software.
5. Under Software Configurations, find Connect to AWS IoT- ST, and then choose Download.
6. Unzip the downloaded file to the AmazonFreeRTOS folder, and make a note of the folder's path.

Note

The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, C:\Users\Username\Dev\AmazonFreeRTOS works, but C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS causes build failures.

In this tutorial, the path to the AmazonFreeRTOS directory is referred to as BASE_FOLDER.

Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. Registering Your MCU Board with AWS IoT (p. 5) is a step in the Prerequisites (p. 4).
To configure your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the navigation pane, choose Settings.
   
   Your AWS IoT endpoint is displayed in Endpoint. It should look like <1234567890123>-ats.iot.<us-east-1>.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose Manage, and then choose Things.
   
   Your device should have an AWS IoT thing name. Make a note of this name.
4. In your IDE, open <BASE_FOLDER>/demos/common/include/aws_clientcredential.h and specify values for the following #define constants:
   
   • clientcredentialMQTT_BROKER_ENDPOINT Your AWS IoT endpoint
   • clientcredentialIOT_THING_NAME The AWS IoT thing name of your board

To configure your Wi-Fi

1. Open the aws_clientcredential.h file.
2. Specify values for the following #define constants:
   
   • clientcredentialWIFI_SSID The SSID for your Wi-Fi network
   • clientcredentialWIFI_PASSWORD The password for your Wi-Fi network
   • clientcredentialWIFI_SECURITY The security type of your Wi-Fi network

   Valid security types are:
   • eWiFiSecurityOpen (Open, no security)
   • eWiFiSecurityWEP (WEP security)
   • eWiFiSecurityWPA (WPA security)
   • eWiFiSecurityWPA2 (WPA2 security)

To configure your AWS IoT credentials

**Note**

To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You must format the certificate and private key for your device.

1. In a browser window, open <BASE_FOLDER>/tools/certificate_configuration/CertificateConfigurator.html.
2. Under Certificate PEM file, choose the <ID>-certificate.pem.crt that you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the <ID>-private.pem.key that you downloaded from the AWS IoT console.
4. Choose Generate and save aws_clientcredential_keys.h, and then save the file in <BASE_FOLDER>/demos/common/include. This overwrites the existing file in the directory.

**Note**

The certificate and private key are hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.
Build and Run Amazon FreeRTOS Samples

Import the Amazon FreeRTOS Sample Code into the STM32 System Workbench

1. Open the STM32 System Workbench and enter a name for a new workspace.
2. From the File menu, choose Import. Expand General, choose Existing Projects into Workspace, and then choose Next.
3. In Select Root Directory, enter `<BASE_FOLDER>\demos\st\stm32l475_discovery\ac6`.
4. The project `aws_demos` should be selected by default.
5. Choose Finish to import the project into STM32 System Workbench.
6. From the Project menu, choose Build All. Confirm the project compiles without any errors or warnings.

Run the Amazon FreeRTOS Samples

1. Use a USB cable to connect your STMicroelectronics STM32L4 Discovery Kit IoT Node to your computer.
2. Rebuild your project.
3. Sign in to the AWS IoT console.
4. In the navigation pane, choose Test to open the MQTT client.
5. In Subscription topic, enter `freertos/demos/echo`, and then choose Subscribe to topic.
6. From Project Explorer, right-click `aws_demos`, choose Debug As, and then choose Ac6 STM32 C/C++ Application.

   If a debug error occurs the first time a debug session is launched, follow these steps:
   1. In STM32 System Workbench, from the Run menu, choose Debug Configurations.
   2. Choose `aws_demos Debug`. (You might need to expand Ac6 STM32 Debugging.)
   3. Choose the Debugger tab.
   4. In Configuration Script, choose Show Generator Options.
   5. In Mode Setup, set Reset Mode to Software System Reset. Choose Apply, and then choose Debug.
   7. When the debugger stops at the breakpoint in `main()`, from the Run menu, choose Resume.

You should see MQTT messages sent by your device in the MQTT client in the AWS IoT console.

Run the Bluetooth Low-Energy Demo

Amazon FreeRTOS support for Bluetooth Low Energy is in public beta release. BLE demos are subject to change.

**Note**
To run the BLE demo, you need the SPBTLE-1S BLE module for the STM32L475 Discovery Kit.

Amazon FreeRTOS supports Bluetooth Low Energy (BLE) connectivity. You can download Amazon FreeRTOS with BLE from GitHub.
For instructions about how to run the MQTT over BLE demo on your board, see the MQTT over BLE demo application.

Troubleshooting

If you see the following in the UART output from the demo application, you need to update the Wi-Fi module’s firmware:

[Tmr Svc] WiFi firmware version is: xxxxxxxxxxxxx
[Tmr Svc] [WARN] WiFi firmware needs to be updated.

To download the latest Wi-Fi firmware, see STM32L4 Discovery kit IoT node, low-power wireless, BLE, NFC, SubGHz, Wi-Fi. In Binary Resources, choose the download link for Inventek ISM 43362 Wi-Fi module firmware update.

Getting Started with the NXP LPC54018 IoT Module

Before you begin, see Prerequisites (p. 4).

If you do not have an NXP LPC54018 IoT Module, you can order one from NXP. Use a USB cable to connect your NXP LPC54018 IoT Module to your computer.

Setting Up Your Environment

Amazon FreeRTOS supports two IDEs for the NXP LPC54018 IoT Module: IAR Embedded Workbench and MCUXpresso.

Before you begin, install one of these IDEs.

To install IAR Embedded Workbench for ARM

1. Browse to Software for NXP Kits and choose Download Software.

   Note
   IAR Embedded Workbench for ARM requires Microsoft Windows.

2. Unzip and run the installer. Follow the prompts.

3. In the License Wizard, choose Register with IAR Systems to get an evaluation license.

To install MCUXpresso from NXP

1. Download and run the MCUXpresso installer from NXP.

2. Browse to MCUXpresso SDK and choose Build your SDK.

3. Choose Select Development Board.


5. Under Boards, choose LPC54018-IoT-Module.

6. Verify the hardware details, and then choose Build MCUXpresso SDK.

7. The SDK for Windows using the MCUXpresso IDE is already built. Choose Download SDK. If you are using another operating system, under Host OS, choose your operating system, and then choose Download SDK.
8. Start the MCUXpresso IDE, and choose the **Installed SDKs** tab.
9. Drag and drop the downloaded SDK archive file into the **Installed SDKs** window.

If you experience issues during installation, see [NXP Support](https://www.nxp.com/support) or [NXP Developer Resources](https://developer.nxp.com).

### Connecting a JTAG Debugger

You need a JTAG debugger to launch and debug your code running on the NXP LPC54018 board. Amazon FreeRTOS was tested using a Segger J-Link probe. For more information about supported debuggers, see the [NXP LPC54018 User Guide](https://www.nxp.com/docs/en/user-guide/LPC54018UG.pdf).

**Note**

If you are using a Segger J-Link debugger, you need a converter cable to connect the 20-pin connector from the debugger to the 10-pin connector on the NXP IoT module.

### Download and Configure Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS.

#### Download Amazon FreeRTOS

1. Browse to the [Amazon FreeRTOS](https://aws.amazon.com/iot-device-software/) page in the AWS IoT console.
2. In the navigation pane, choose **Software**.
3. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
4. Choose **Download FreeRTOS Software**.
5. Under **Software Configurations**, find **Connect to AWS IoT- NXP**, and then:
   - If you are using IAR Workbench, choose **Download**.
   - If you are using MCUXpresso:
     a. In **Software Configurations**, find **Connect to AWS IoT- NXP**. Select **Connect to AWS IoT- NXP**, but do not choose **Download**.
     b. Under **Hardware Platform**, choose **Edit**.
     c. Under **Integrated Development Environment (IDE)**, choose **MCUXpresso**.
     d. Under **Compiler**, choose **GCC**.
     e. At the bottom of the page, choose **Create and Download**.
6. Unzip the downloaded file to the AmazonFreeRTOS folder and make a note of the folder's path.

**Note**

The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, `C:\Users\Username\Dev\AmazonFreeRTOS` works, but `C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` causes build failures.

In this tutorial, the path to the AmazonFreeRTOS directory is referred to as **BASE_FOLDER**.

#### Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. [Registering Your MCU Board with AWS IoT (p. 5)](https://aws.amazon.com/iot-device-software/) is a step in the **Prerequisites (p. 4)**.
To configure your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the navigation pane, choose Settings.
   
   Your AWS IoT endpoint is displayed in Endpoint. It should look like <1234567890123>-ats.iot.<us-east-1>.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose Manage, and then choose Things.
   
   Your device should have an AWS IoT thing name. Make a note of this name.
4. In your IDE, open `<BASE_FOLDER>\demos\common\include\aws_clientcredential.h` and specify values for the following #define constants:

   • clientcredentialMQTT_BROKER_ENDPOINT Your AWS IoT endpoint
   • clientcredentialIOT_THING_NAME The AWS IoT thing name of your board

To configure your Wi-Fi

1. Open the `aws_clientcredential.h` file.
2. Specify values for the following #define constants:

   • clientcredentialWIFI_SSID The SSID for your Wi-Fi network
   • clientcredentialWIFI_PASSWORD The password for your Wi-Fi network
   • clientcredentialWIFI_SECURITY The security type of your Wi-Fi network

   Valid security types are:
   • eWiFiSecurityOpen (Open, no security)
   • eWiFiSecurityWEP (WEP security)
   • eWiFiSecurityWPA (WPA security)
   • eWiFiSecurityWPA2 (WPA2 security)

To configure your AWS IoT credentials

**Note**
To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You must format the certificate and private key for your device.

1. In a browser window, open `<BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html`.
2. Under Certificate PEM file, choose the `<ID>-certificate.pem.crt` that you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the `<ID>-private.pem.key` that you downloaded from the AWS IoT console.
4. Choose Generate and save aws_clientcredential_keys.h, and then save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the existing file in the directory.

**Note**
The certificate and private key are hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.
Build and Run Amazon FreeRTOS Samples

Import the Amazon FreeRTOS Sample Code into Your IDE

To import the Amazon FreeRTOS sample code into the IAR Embedded Workbench IDE

1. Open IAR Embedded Workbench, and from the File menu, choose Open Workspace.
2. In the search-directory text box, enter \demos\nxp\lpc54018_iot_module \iar, and choose aws_demos.eww.
3. From the Project menu, choose Rebuild All.

To import the Amazon FreeRTOS sample code into the MCUXpresso IDE

1. Open MCUXpresso, and from the File menu, choose Open Projects From File System.
2. In the Directory text box, enter \demos\nxp\lpc54018_iot_module \mcuxpresso, and choose Finish.
3. From the Project menu, choose Build All.

Run the FreeRTOS Samples

To run the Amazon FreeRTOS demos on the NXP LPC54018 IoT Module board, connect the USB port on the NXP IoT Module to your host computer, open a terminal program, and connect to the port identified as USB Serial Device.

1. Rebuild your project.
2. Sign in to the AWS IoT console.
3. In the navigation pane, choose Test to open the MQTT client.
4. In Subscription topic, enter freertos/demos/echo, and then choose Subscribe to topic.
5. In your IDE, from the Project menu, choose Build.
6. Connect the NXP IoT Module and the Segger J-Link Debugger to the USB ports on your computer using mini-USB to USB cables.
7. If you are using IAR Embedded Workbench:
   a. From the Project menu, choose Download and Debug.
   b. From the Debug menu, choose Start Debugging.
   c. When the debugger stops at the breakpoint in main, from the Debug menu, choose Go.

   Note
   If a J-Link Device Selection dialog box opens, choose OK to continue. In the Target Device Settings dialog box, choose Unspecified, choose Cortex-M4, and then choose OK. You only need to do this once.
8. If you are using MCUXpresso:
   a. If this is your first time debugging, choose the aws_demos project and from the Debug toolbar, choose the blue debug button.
   b. Any detected debug probes are displayed. Choose the probe you want to use, and then choose OK to start debugging.
Note
When the debugger stops at the breakpoint in `main()`, press the debug restart button once to reset the debugging session. (This is required due to a bug with MCUXpresso debugger for NXP54018-IoT-Module).

9. When the debugger stops at the breakpoint in `main()`, from the Debug menu, choose Go.

You should see MQTT messages sent by your device in the MQTT client of the AWS IoT console.

Troubleshooting

If no messages appear in the AWS IoT console, try the following:

1. Open a terminal window to view the logging output of the sample. This can help you determine what is going wrong.
2. Check that your network credentials are valid.

Getting Started with the Microchip Curiosity PIC32MZEF

Before you begin, see Prerequisites (p. 4).

If you do not have the Microchip Curiosity PIC32MZEF bundle, you can purchase one from Microchip. You need the following items:

- MikroElectronika USB UART Click Board
- RJ-11 to ICSP Adapter
- MPLAB ICD 4 In-Circuit Debugger
- PIC32 LAN8720 PHY daughter board
- MikroElectronika WiFi 7 Click Board

Setting Up the Microchip Curiosity PIC32MZEF Hardware

1. Connect the MikroElectronika USB UART Click Board to the microBUS 1 connector on the Microchip Curiosity PIC32MZEF.
2. Connect the PIC32 LAN8720 PHY daughter board to the J18 header on the Microchip Curiosity PIC32MZEF.
3. Connect the MikroElectronika USB UART Click Board to your computer using a USB A to USB mini-B cable.
4. Connect the MikroElectronika WiFi 7 Click Board to the microBUS 2 connector on the Microchip Curiosity PIC32MZEF.
5. Connect the RJ-11 to ICSP Adapter to the Microchip Curiosity PIC32MZEF.
6. Connect the MPLAB ICD 4 In-Circuit Debugger to your Microchip Curiosity PIC32MZEF using an RJ-11 cable.
7. Connect the ICD 4 In-Circuit Debugger to your computer using a USB A to USB mini-B cable.
8. Insert the RJ-11 to ICSP Adaptor J2 into the ICSP header on the Microchip Curiosity PIC32MZEF at the J16.
9. Connect one end of an Ethernet cable to the LAN8720 PHY daughter board. Connect the other end to your router or other internet port.

The following image shows the Microchip Curiosity PIC32MZEF and all required peripherals assembled.

The LED on in-circuit debugger turns a solid blue when it is ready.

Setting Up Your Environment

1. Install the latest Java SE SDK.
2. Install Python version 3.x or later.
3. Install the latest version of the MPLAB X IDE:
   - MPLAB X Integrated Development Environment for Windows
   - MPLAB X Integrated Development Environment for macOS
   - MPLAB X Integrated Development Environment for Linux
4. Install the latest version of the MPLAB XC32 Compiler:
   - MPLAB XC32/32++ Compiler for Windows
   - MPLAB XC32/32++ Compiler for macOS
   - MPLAB XC32/32++ Compiler for Linux
5. Install the latest version of the MPLAB Harmony Integrated Software Framework (optional):
   - MPLAB Harmony Integrated Software Framework for Windows
   - MPLAB Harmony Integrated Software Framework for macOS
   - MPLAB Harmony Integrated Software Framework for Linux

6. Start up a UART terminal emulator and open a connection with the following settings:
   - Baud rate: 115200
   - Data: 8 bit
   - Parity: None
   - Stop bits: 1
   - Flow control: None

Download and Configure Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS.

Download Amazon FreeRTOS

1. Browse to the Amazon FreeRTOS page in the AWS IoT console.
2. In the navigation pane, choose Software.
4. Choose Download FreeRTOS Software.
5. In Software Configurations, find Connect to AWS IoT- Microchip, and then choose Download.
6. Unzip the downloaded file to the AmazonFreeRTOS folder, and make a note of the folder's path.

Note
The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, C:\Users\Username\Dev\AmazonFreeRTOS works, but C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS causes build failures.
In this tutorial, the path to the AmazonFreeRTOS directory is referred to as BASE_FOLDER.

Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. Registering Your MCU Board with AWS IoT (p. 5) is a step in the Prerequisites (p. 4).

To configure your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the navigation pane, choose Settings.
   - Your AWS IoT endpoint is displayed in Endpoint. It should look like <1234567890123>-ats.iot.<us-east-1>.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose Manage, and then choose Things.
   - Your device should have an AWS IoT thing name. Make a note of this name.
4. In your IDE, open `<BASE_FOLDER>`\demos\common\include\aws_clientcredential.h and specify values for the following `#define` constants:

- `clientcredentialMQTT_BROKER_ENDPOINT` *Your AWS IoT endpoint*
- `clientcredentialIOT_THING_NAME` *The AWS IoT thing name of your board*

**To configure your Wi-Fi**

1. Open the `aws_clientcredential.h` file.
2. Specify values for the following `#define` constants:

   - `clientcredentialWIFI_SSID` *The SSID for your Wi-Fi network*
   - `clientcredentialWIFI_PASSWORD` *The password for your Wi-Fi network*
   - `clientcredentialWIFI_SECURITY` *The security type of your Wi-Fi network*

   Valid security types are:
   - `eWiFiSecurityOpen` (Open, no security)
   - `eWiFiSecurityWEP` (WEP security)
   - `eWiFiSecurityWPA` (WPA security)
   - `eWiFiSecurityWPA2` (WPA2 security)

**To configure your AWS IoT credentials**

**Note**

To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You must format the certificate and private key for your device.

1. In a browser window, open `<BASE_FOLDER>`\tools\certificate_configuration\CertificateConfigurator.html.
2. Under **Certificate PEM file**, choose the `<ID>-certificate.pem.crt` that you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the `<ID>-private.pem.key` that you downloaded from the AWS IoT console.
4. Choose **Generate and save aws_clientcredential_keys.h**, and then save the file in `<BASE_FOLDER>`\demos\common\include. This overwrites the existing file in the directory.

**Note**

The certificate and private key are hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.

**Build and Run Amazon FreeRTOS Samples**

**Open the Amazon FreeRTOS Demo Application in the MPLAB IDE**

1. In the MPLAB IDE, from the **File** menu, choose **Open Project**.
2. Browse to and open `<BASE_FOLDER>`\demos\microchip\curiosity_pic32mzef\mplab.
3. Choose **Open project**.

**Note**
When you open the project for the first time, you can ignore warning messages like the following:

```
warning: Configuration "pic32mz_ef_curiosity" builds with "XC32", but indicates no toolchain directory.
warning: Configuration "pic32mz_ef_curiosity" refers to file "AmazonFreeRTOS/lib/third_party/mcu_vendor/microchip/harmony/framework/bootloader/src/bootloader.h"
which does not exist in the disk. The make process might not build correctly.
```

---

## Run the Amazon FreeRTOS Samples

1. Rebuild your project.
2. Sign in to the **AWS IoT console**.
3. In the navigation pane, choose **Test** to open the MQTT client.
4. In **Subscription topic**, enter **freertos/demos/echo**, and then choose **Subscribe to topic**.
5. On the **Projects** tab, right-click the **aws_demos** top-level folder, and then choose **Debug**.
6. The first time you debug the samples, an **ICD 4 not Found** dialog box is displayed. In the tree view, under the **ICD 4** node, choose the ICD4 serial number, and then choose **OK**.
7. When the debugger stops at the breakpoint in **main()**, from the **Run** menu, choose **Resume**.

The ICD 4 turns half yellow as it is programming the device, and then half green when it is running. The **ICD4** tab appears in the IDE. Successful programming looks like the following:

```
*****************************************************
Connecting to MPLAB ICD 4...
Currently loaded versions:
Application version............01.02.00
Boot version...................01.00.00
FPGA version...................01.00.00
Script version.................00.02.18
Script build number............fd44437f19
Application build number.......0123456789
Connecting to MPLAB ICD 4...
Currently loaded versions:
Boot version...................01.00.00
Updating firmware application...
Connecting to MPLAB ICD 4...
Currently loaded versions:
Application version............01.02.16
Boot version...................01.00.00
FPGA version...................01.00.00
Script version.................00.02.18
Script build number............fd44437f19
Application build number.......0123456789
Target voltage detected
Target device PIC32MZ2048EFM100 found.
```
Device Id Revision = 0xA1
Serial Number:
  Num0 = ec4f6d3c
  Num1 = 6b845410

Erasing...

The following memory area(s) will be programmed:
- program memory: start address = 0x1d000000, end address = 0x1d07bfff
- program memory: start address = 0x1d1fc000, end address = 0x1d1fffff
- configuration memory
- boot config memory

Programming/Verify complete
Running

**Note**

We recommend that you use the MPLAB In-Circuit Debugger instead of the USB port for debugging. The ICD 4 makes it possible for you to step through code more quickly and add breakpoints without restarting the debugger.

You should see MQTT messages sent by your device in the MQTT client of the AWS IoT console.

**Troubleshooting**

If no messages appear in the AWS IoT console, try the following:

1. Open a terminal window to view the logging output of the sample. This can help you determine what is going wrong.
2. Check that your network credentials are valid.

**Getting Started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT**

Both the ESP32-DevKitC and the ESP-WROVER KIT are supported on Amazon FreeRTOS. To check which development module you have, see ESP32 Modules and Boards.

**Note**

Currently, the Amazon FreeRTOS port for ESP32-WROVER-KIT and ESP DevKitC does not support the following:

- Lightweight IP.
- Symmetric multiprocessing (SMP).

**Setting Up the Espressif Hardware**

For the ESP32-DevKitC development board, see Getting Started with the ESP32-DevKitC development board.

For the ESP-WROVER-KIT development board, see Getting Started with the ESP-WROVER-KIT development board.
Setting Up Your Environment

Establishing a Serial Connection

To establish a serial connection with the ESP32-DevKitC, you must install CP210x USB to UART Bridge VCP drivers. If you are running Windows 10, install v6.7.5 of the CP210x USB to UART Bridge drivers. If you are running an older version of Windows, install the version indicated in the list of downloads.

To establish a serial connection with the ESP32-WROVER-KIT, you must install some FTDI virtual COM port drivers. For more information, see Establishing a Serial Connection with ESP32.

Make a note of the serial port you configure (based on host OS). You need it during the build process.

Setting Up the Toolchain

When following the links below, do not install the ESP-IDF library from Espressif, Amazon FreeRTOS already contains this library. In addition, make sure the \texttt{IDF\_PATH} environment variable has not been set.

- Setting up the toolchain for Windows
- Setting up the toolchain for macOS
- Setting up the toolchain for Linux

Download and Configure Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS.

Downloading Amazon FreeRTOS

Clone the Amazon FreeRTOS repository from GitHub.

\textbf{Note}

The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, \texttt{C:\Users\Username\Dev\AmazonFreeRTOS} works, but \texttt{C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS} causes build failures.

In this tutorial, the path to the AmazonFreeRTOS directory is referred to as \texttt{BASE\_FOLDER}.

Configure Your Project

1. If you are running macOS or Linux, open a terminal prompt. If you are running Windows, open \texttt{mingw32.exe}.
2. Install Python 2.7.10 or later.
3. If you are running Windows, use the \texttt{easy\_install awscli} to install the AWS CLI in the \texttt{mingw32} environment.

   If you are running macOS or Linux, make sure the AWS CLI is installed on your system. For more information, see \texttt{Installing the AWS Command Line Interface}.
4. Run \texttt{aws configure} to configure the AWS CLI. For more information, see \texttt{Configuring the AWS CLI}.
5. Use the following command to install the boto3 Python module:
Amazon FreeRTOS includes scripts to make it easier to set up your Espressif board. To configure the Espressif scripts, open `<BASE_FOLDER>/tools/aws_config_quick_start/configure.json` and set the following attributes:

- `afr_source_dir`
  
  The complete path to the Amazon FreeRTOS download on your computer.

- `thing_name`
  
  The name of the IoT thing that represents your board.

- `wifi_ssid`
  
  The SSID of your Wi-Fi network.

- `wifi_password`
  
  The password for your Wi-Fi network.

- `wifi_security`
  
  The security type for your Wi-Fi network.

  Valid security types are:
  - `eWiFiSecurityOpen` (Open, no security)
  - `eWiFiSecurityWEP` (WEP security)
  - `eWiFiSecurityWPA` (WPA security)
  - `eWiFiSecurityWPA2` (WPA2 security)

To run the configuration script

1. If you are running macOS or Linux, open a terminal prompt. If you are running Windows, open mingw32.exe.
2. Go to the `<BASE_FOLDER>/tools/aws_config_quick_start` directory and run the following command:

   ```
   python SetupAWS.py setup
   ```

This script creates an IoT thing, certificate, and policy. It attaches the IoT policy to the certificate and the certificate to the IoT thing. It also populates the `aws_clientcredential.h` file with your AWS IoT endpoint, Wi-Fi SSID, and credentials. Finally, it formats your certificate and private key and writes them to the `aws_clientcredential.h` header file. For more information about the script, see the README.md in the `<BASE_FOLDER>/tools/aws_config_quick_start` directory.

## Build and Run Amazon FreeRTOS Samples

### To flash the demo application onto your board

1. Connect your board to your computer.
2. In macOS or Linux, open a terminal. In Windows, open mingw32.exe (downloaded from mysys toolchain).
3. Navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` and enter following command:

```
make menuconfig
```

In the Espressif IoT Development Framework Configuration menu, navigate to **Serial flasher config**, and then to **Default serial port** to configure the serial port.

On Windows, serial ports have names like `COM1`. On macOS, they start with `/dev/cu`. On Linux, they start with `/dev/tty`.

The serial port you configure here is used to write the demo application to your board.

Depending on your hardware, you can increase the default baud rate up to 921600. This can reduce the time required to flash your board. To increase the baud rate, choose **Serial flash config**, and then choose **Default baud rate**.

To confirm your selection, choose **ENTER**. To save the configuration, choose **Save** and then choose **Exit**.

To build and flash firmware (including boot loader and partition table) and monitor serial console output, open a command prompt. Navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` and run the following command:

```
make flash monitor
```

At the end of the compilation output, you should see text like the following:

```
I (31) boot: ESP-IDF v3.1-dev-322-gf307f41-dirty 2nd stage bootloader
I (31) boot: compile time 11:30:50
I (34) boot: Enabling RNG early entropy source...
I (37) boot: SPI Speed : 40MHz
I (41) boot: SPI Mode : DIO
I (45) boot: SPI Flash Size : 4MB
I (49) boot: Partition Table:
I (53) boot: ## Label            Usage          Type ST Offset   Length
I (60) boot:  0 nvs              WiFi data        01 02 00009000 00006000
I (68) boot:  1 phy_init         RF data          01 01 0000f000 00001000
I (75) boot:  2 factory         factory app      00 00 00010000 00100000
I (82) boot:  3 storage          Unknown data     01 82 00110000 00010000
I (90) boot: End of partition table
I (94) esp_image: segment 0: paddr=0x00010020 vaddr=0x3f400020 size=0x12710 ( 75536) map
I (129) esp_image: segment 1: paddr=0x000022738 vaddr=0x3ff00000 size=0x0240c ( 9228) load
I (133) esp_image: segment 2: paddr=0x000024b4c vaddr=0x40080000 size=0x00400 ( 1024) load
0x40080000: _iram_start at BASE_FOLDER/AmazonFreeRTOS-Espressif/lib/FreeRTOS/portable/GCC/Xtensa_ESP32/xtensa_vectors.S:1685
I (136) esp_image: segment 3: paddr=0x000024f54 vaddr=0x40080400 size=0x0b0bc ( 45244) load
I (164) esp_image: segment 4: paddr=0x000030018 vaddr=0x400d0018 size=0x6d454 (447572) map
0x400d0018: _stext at ??:?
I (319) esp_image: segment 5: paddr=0x00009d474 vaddr=0x4008b4bc size=0x02d44 (11588) load
0x4008b4bc: xStreamBufferSend at BASE_FOLDER/AmazonFreeRTOS-Espressif/lib/FreeRTOS/stream_buffer.c:636
I (324) esp_image: segment 6: paddr=0x0000a01c0 vaddr=0x400c0000 size=0x00000 ( 0) load
I (334) boot: Loaded app from partition at offset 0x10000
I (334) boot: Disabling RNG early entropy source...
I (338) cpu_start: Pro cpu up.
```
I (341) cpu_start: Single core mode
I (346) heap_init: Initializing. RAM available for dynamic allocation:
I (353) heap_init: At 3FFA6E00 len 00001920 (6 KiB): DRAM
I (359) heap_init: At 3FFC0420 len 0001FB0 (126 KiB): DRAM
I (365) heap_init: At 3FFE0440 len 0003BC0 (14 KiB): D/IRAM
I (371) heap_init: At 3FFE4350 len 0001B0 (111 KiB): D/IRAM
I (378) heap_init: At 4008E200 len 0011E00 (71 KiB): IRAM
I (384) cpu_start: Pro cpu start user code
I (66) cpu_start: Starting scheduler on PRO CPU.
I (96) wifi: wifi firmware version: f79168c
I (96) wifi: config NVS flash: enabled
I (96) wifi: config nano formatting: disabled
I (106) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (106) system_api: Base MAC address is not set, read default base MAC address from BLK0 of EFUSE
I (136) wifi: Init dynamic tx buffer num: 32
I (136) wifi: Init data frame dynamic rx buffer num: 32
I (136) wifi: Init management frame dynamic rx buffer num: 32
I (136) wifi: wifi driver task: 3ffcc4e4, prio: 23, stack: 4096
I (146) wifi: Init static rx buffer num: 10
I (146) wifi: Init dynamic rx buffer num: 32
I (156) wifi: wifi power manager task: 0x3ffcc248 prio: 21 stack: 2560
0 7 [Tmr Svc] WiFi module initialized. Connecting to AP Guest...
W (166) phy_init: failed to load RF calibration data (0x1102), falling back to full calibration
I (396) phy: phy_version: 383.0, 79a622c, Jan 30 2018, 15:38:06, 0, 2
I (406) wifi: mode : sta (30:ae:a4:4b:3d:64)
I (406) WIFI: SYSTEM_EVENT_STA_START
I (526) wifi: n:1 0, o:1 0, ap:255 255, sta:1 0, prof:1
I (526) wifi: state: init -> auth (b0)
I (536) wifi: state: auth -> assoc (0)
I (536) wifi: state: assoc -> run (10)
I (536) wifi: connected with Guest, channel 1
I (536) WIFI: SYSTEM_EVENT_STA_CONNECTED
I (3536) wifi: pm start, type:0
1 826 [IP-task] vDHCPProcess: offer c0a8520bip
I (8356) event: sta ip: 192.168.82.11, mask: 255.255.224.0, gw: 192.168.64.1
I (8356) WIFI: SYSTEM_EVENT_STA_GOT_IP
2 827 [IP-task] vDHCPProcess: offer c0a8520bip
3 828 [Tmr Svc] WiFi Connected to AP. Creating tasks which use network...
4 828 [Tmr Svc] Creating MQTT Echo Task...
5 829 [MQTTEcho] MQTT echo attempting to connect to a1405vz6c0ikzv.iot.us-west-2.amazonaws.com.
6 829 [MQTTEcho] Sending command to MQTT task.
7 830 [MQTT] Received message 10000 from queue.
8 2030 [IP-task] Socket sending wakeup to MQTT task.
I (20416) PKCS11: Initializing SPIFFS
9 2596 [MQTT] Received message 0 from queue.
10 2601 [IP-task] Socket sending wakeup to MQTT task.
11 2601 [MQTT] Received message 0 from queue.
12 2607 [IP-task] Socket sending wakeup to MQTT task.
13 2607 [MQTT] Received message 0 from queue.
14 2607 [MQTT] MQTT Connect was accepted. Connection established.
15 2607 [MQTT] Notifying task.
16 2608 [MQTTEcho] Command sent to MQTT task passed.
17 2608 [MQTTEcho] MQTT echo connected.
18 2608 [MQTTEcho] MQTT echo test echoing task created.
19 2608 [MQTTEcho] Sending command to MQTT task.
20 2609 [MQTT] Received message 20000 from queue.
21 2610 [IP-task] Socket sending wakeup to MQTT task.
22 2611 [MQTT] Received message 0 from queue.
23 2612 [IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[MQTT] MQTT Subscribe was accepted. Subscribed.
[MQTT] Notifying task.
[MQTT] Command sent to MQTT task passed.
MQTT Echo demo subscribed to freertos/demos/echo
[MQTT] Sending command to MQTT task.
[MQTT] Received message 30000 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] MQTT Publish was successful.
[MQTT] Notifying task.
[MQTT] Command sent to MQTT task passed.
[MQTT] Echo successfully published 'Hello World 0'
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 40000 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[MQTT] MQTT Publish was successful.
[MQTT] Notifying task.
[Echoing] Command sent to MQTT task passed.
*** Similar output deleted for brevity ***
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 190000 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] MQTT Publish was successful.
[MQTT] Notifying task.
[Echoing] Command sent to MQTT task passed.
[MQTT] Command sent to MQTT task passed.
[MQTT] Echo successfully published 'Hello World 11'
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 1a0000 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] Received message 0 from queue.
[IP-task] Socket sending wakeup to MQTT task.
[MQTT] MQTT Publish was successful.
[MQTT] Notifying task.
[MQTT] Command sent to MQTT task passed.
[MQTT] Received message 1a0000 from queue.
[IP-task] Socket sending wakeup to MQTT task.
Run the Bluetooth Low-Energy Demos

Amazon FreeRTOS support for Bluetooth Low Energy is in public beta release. BLE demos are subject to change.

Amazon FreeRTOS supports Bluetooth Low Energy (BLE) connectivity. You can download Amazon FreeRTOS with BLE from GitHub.

For instructions about how to run the MQTT over BLE demo on your board, see the MQTT over BLE demo application.

For instructions about how to run the Wi-Fi Provisioning demo on your board, see the Wi-Fi Provisioning demo application.

Troubleshooting

- If you are using a Mac and it does not recognize your ESP-WROVER-KIT, make sure you do not have the D2XX drivers installed. To uninstall them, follow the instructions in the FTDI Drivers Installation Guide for macOS X.
- The monitor utility provided by ESP-IDF (and invoked using make monitor) helps you decode addresses. For this reason, it can help you get some meaningful backtraces in the event the application crashes. For more information, see Automatically Decoding Addresses on the Espressif website.
- It is also possible to enable GDBstub for communication with gdb without requiring any special JTAG hardware. For more information, see Launch GDB for GDBStub.
- For information about setting up an OpenOCD-based environment if JTAG hardware-based debugging is required, see JTAG Debugging.
- If pyserial cannot be installed using pip on macOS, download it from pyserial.
- If the board resets continuously, try erasing the flash by entering the following command on the terminal:

  ```
  make erase_flash
  ```

- If you see errors when you run idf_monitor.py, use Python 2.7.

Other Notes

- Required libraries from ESP-IDF are included in Amazon FreeRTOS, so there is no need to download them externally. If IDF_PATH is set, we recommend that you remove it before you build Amazon FreeRTOS.
- On Window systems, it can take 3-4 minutes for the project to build. You can use the –j4 switch on the make command to reduce the build time:

  ```
  make flash monitor -j4
  ```
Debugging Code on Espressif ESP32-DevKitC and ESP-WROVER-KIT

You need a JTAG to USB cable. We use a USB to MPSSE cable (for example, the FTDI C232HM-DDHSL-0).

ESP-DevKitC JTAG Setup

For the FTDI C232HM-DDHSL-0 cable, these are the connections to the ESP32 DevkitC:

<table>
<thead>
<tr>
<th>C232HM-DDHSL-0 Wire Color</th>
<th>ESP32 GPIO Pin</th>
<th>JTAG Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown (pin 5)</td>
<td>IO14</td>
<td>TMS</td>
</tr>
<tr>
<td>Yellow (pin 3)</td>
<td>IO12</td>
<td>TDI</td>
</tr>
<tr>
<td>Black (pin 10)</td>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>Orange (pin 2)</td>
<td>IO13</td>
<td>TCK</td>
</tr>
<tr>
<td>Green (pin 4)</td>
<td>IO15</td>
<td>TDO</td>
</tr>
</tbody>
</table>

ESP-WROVER-KIT JTAG Setup

For the FTDI C232HM-DDHSL-0 cable, these are the connections to the ESP32-WROVER-KIT:

<table>
<thead>
<tr>
<th>C232HM-DDHSL-0 Wire Color</th>
<th>ESP32 GPIO Pin</th>
<th>JTAG Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown (pin 5)</td>
<td>IO14</td>
<td>TMS</td>
</tr>
<tr>
<td>Yellow (pin 3)</td>
<td>IO12</td>
<td>TDI</td>
</tr>
<tr>
<td>Orange (pin 2)</td>
<td>IO13</td>
<td>TCK</td>
</tr>
<tr>
<td>Green (pin 4)</td>
<td>IO15</td>
<td>TDO</td>
</tr>
</tbody>
</table>

These tables were developed from the FTDI C232HM-DDHSL-0 datasheet. For more information, see C232HM MPSSE Cable Connection and Mechanical Details in the datasheet.

To enable JTAG on the ESP-WROVER-KIT, place jumpers on the TMS, TDO, TDI, TCK, and S_TDI pins as shown here:
### Debugging on Windows

**To set up for debugging on Windows**

1. Connect the USB side of the FTDI C232HM-DDHSL-0 to your computer and the other side as described in Debugging Code on Espressif ESP32-DevKitC and ESP-WROVER-KIT (p. 32). The FTDI C232HM-DDHSL-0 device should appear in **Device Manager** under **Universal Serial Bus Controllers**.

2. From the list of USB controllers, right-click the FTDI C232HM-DDHSL-0 device (the manufacturer is FTDI), and choose **Properties**. In the properties window, choose the **Details** tab to see the properties of the device. If the device is not listed, install the Windows driver for FTDI C232HM-DDHSL-0.

3. Verify that the vendor ID and product ID displayed in **Device Manager** match the IDs in `demos\espressif\esp32_devkitc_esp_wrover_kit\esp32_devkitj_v1.cfg`. The IDs are specified in a line that begins with `ftdi_vid_pid` followed by a vendor ID and a product ID:

   ```
   ftdi_vid_pid 0x0403 0x6014
   ```

4. Download OpenOCD for Windows.

5. Unzip the file to `C:` and add `C:\openocd-esp32\bin` to your system path.

6. OpenOCD requires libusb, which is not installed by default on Windows. To install it:
   a. Download zadig.exe.
   b. Run zadig.exe. From the **Options** menu, choose **List All Devices**.
   c. From the drop-down menu, choose **C232HM-DDHSL-0**.
   d. In the target driver box, to the right of the green arrow, choose **WinUSB**.
   e. From the drop-down box under the target driver box, choose the arrow, and then choose **Install Driver**. Choose **Replace Driver**.

7. Open a command prompt, navigate to `<BASE_FOLDER>\demos\espressif\esp32_devkitc_esp_wrover_kit\make` and run:

   ```
   openocd.exe -f esp32_devkitj_v1.cfg -f esp-wroom-32.cfg
   ```

   Leave this command prompt open.

8. Open a new command prompt, navigate to your `msys32` directory, and run `mingw32.exe`.

   In the mingw32 terminal, navigate to `<BASE_FOLDER>\demos\espressif\esp32_devkitc_esp_wrover_kit\make` and run **make flash monitor**.
9. Open another mingw32 terminal, navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` and run `xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf`. The program should stop in the main function.

**Note**
The ESP32 supports a maximum of two break points.

### Debugging on macOS

1. Download the FTDI driver for macOS.
2. Download OpenOCD.
3. Extract the downloaded .tar file and set the path in `.bash_profile` to `<OCD_INSTALL_DIR>/openocd-esp32/bin`.
4. Use the following command to install `libusb` on macOS:
   ```bash
   brew install libusb
   ```
5. Use the following command to unload the serial port driver:
   ```bash
   sudo kextunload -b com.FTDI.driver.FTDIUSBSerialDriver
   ```
6. If you are running a macOS version later than 10.9, use the following command to unload Apple's FTDI driver:
   ```bash
   sudo kextunload -b com.apple.driver.AppleUSBFTDI
   ```
7. Use the following command to get the product ID and vendor ID of the FTDI cable. It lists the attached USB devices:
   ```bash
   system_profiler SPUSBDataType
   ```
   The output from `system_profiler` should look like the following:

   ```text
   C232HM-DDHSL-0:
   Product ID: 0x6014
   Vendor ID: 0x0403 (Future Technology Devices International Limited)
   ```
8. Verify the vendor and product IDs match the IDs in `demos/espressif/esp32_devkitc_esp_wrover_kit/esp32_devkitj_v1.cfg`. The IDs are specified on a line that begins with `ftdi_vid_pid` followed by a vendor ID and a product ID:
   ```text
   ftdi_vid_pid 0x0403 0x6014
   ```

9. Open a terminal window, navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and use the following command to run OpenOCD:
   ```bash
   openocd -f esp32_devkitj_v1.cfg -f esp-wroom-32.cfg
   ```
10. Open a new terminal, and use the following command to load the FTDI serial port driver:
    ```bash
    sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
    ```
11. Navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and run the following command:
make flash monitor

12. Open another new terminal, navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and run the following command:

   `xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf`

   The program should stop at `main()`.

**Debugging on Linux**

1. Download OpenOCD. Extract the tarball and follow the installation instructions in the readme file.
2. Use the following command to install `libusb` on Linux:

   `sudo apt-get install libusb-1.0`

3. Open a terminal and enter `ls -l /dev/ttyUSB*` to list all USB devices connected to your computer. This helps you check if the board's USB ports are recognized by the operating system. You should see output similar to the following:

   ```
   $ ls -l /dev/ttyUSB*
   crw-rw---- 1 root dialout 188, 0 Jul 10 19:04 /dev/ttyUSB0
   crw-rw---- 1 root dialout 188, 1 Jul 10 19:04 /dev/ttyUSB1
   ```

4. Sign off and then sign in and cycle the power to the board to make the changes take effect. In a terminal prompt, list the USB devices. Make sure the group-owner has changed from `dialout` to `plugdev`:

   ```
   $ ls -l /dev/ttyUSB*
   crw-rw---- 1 root plugdev 188, 0 Jul 10 19:04 /dev/ttyUSB0
   crw-rw---- 1 root plugdev 188, 1 Jul 10 19:04 /dev/ttyUSB1
   ```

   The `/dev/ttyUSBn` interface with the lower number is used for JTAG communication. The other interface is routed to the ESP32's serial port (UART) and is used for uploading code to the ESP32's flash memory.

5. In a terminal window, navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and use the following command to run OpenOCD:

   `openocd -f esp32_devkitj_v1.cfg -f esp-wroom-32.cfg`

6. Open another terminal, navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and run the following command:

   `make flash monitor`

7. Open another terminal, navigate to `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and run the following command:

   `xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf`

   The program should stop in `main()`.
Getting Started with the Infineon XMC4800 IoT Connectivity Kit

Before you begin, see Prerequisites (p. 4).

If you do not have the Infineon XMC4800 IoT Connectivity Kit, you can purchase one from Infineon.

If you want to open a serial connection with the board to view logging and debugging information, you need a 3.3V USB/Serial converter, in addition to the XMC4800 IoT Connectivity Kit. The CP2104 is a common USB/Serial converter that is widely available in boards such as Adafruit's CP2104 Friend.

Setting Up Your Environment

Amazon FreeRTOS uses Infineon's DAVE development environment to program the XMC4800. Before you begin, you need to download and install DAVE and some J-Link drivers to communicate with the on-board debugger.

Install DAVE

1. Go to Infineon's DAVE software download page.
2. Choose the DAVE package for your operating system and submit your registration information. After registering with Infineon, you should receive a confirmation email with a link to download a .zip file.
3. Download the DAVE package .zip file (DAVE_version_os_date.zip), and unzip it to the location where you want to install DAVE (for example, C:\DAVE4).
   
   **Note**
   Some Windows users have reported problems using Windows Explorer to unzip the file. We recommend that you use a third-party program such as 7-Zip.

4. To launch DAVE, run the executable file found in the unzipped DAVE_version_os_date.zip folder.

For more information, see the DAVE Quick Start Guide.

Install Segger J-Link Drivers

To communicate with the XMC4800 Relax EtherCAT board's on-board debugging probe, you need the drivers included in the J-Link Software and Documentation pack. You can download the J-Link Software and Documentation pack from Segger's J-Link software download page.

Set Up a Serial Connection

Setting up a serial connection is optional, but recommended. A serial connection allows your board to send logging and debugging information in a form that you can view on your development machine.

The XMC4800 demo application uses a UART serial connection on pins P0.0 and P0.1, which are labeled on the XMC4800 Relax EtherCAT board's silkscreen. To set up a serial connection:

1. Connect the pin labeled "RX<P0.0" to your USB/Serial converter's "TX" pin.
2. Connect the pin labeled "TX>P0.1" to your USB/Serial converter's "RX" pin.
3. Connect your serial converter's Ground pin to one of the pins labeled "GND" on your board. The devices must share a common ground.
Power is supplied from the USB debugging port, so do not connect your serial adapter's positive voltage pin to the board.

**Note**

Some serial cables use a 5V signaling level. The XMC4800 board and the Wi-Fi Click module require a 3.3V. Do not use the board's IOREF jumper to change the board's signals to 5V.

With the cable connected, you can open a serial connection on a terminal emulator such as GNU Screen. The baud rate is set to 115200 by default with 8 data bits, no parity, and 1 stop bit.

### Download and Configure Amazon FreeRTOS

After you set up your environment, you can download Amazon FreeRTOS.

#### Download Amazon FreeRTOS

1. Browse to the AWS IoT console.
2. In the navigation pane, choose **Software**.
3. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
4. Under **Software Configurations**, find **Connect to AWS IoT- Infineon**, and then choose **Download**.
5. Unzip the downloaded file to the AmazonFreeRTOS folder, and make note of the folder path.

**Note**

The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, `C:\Users\Username\Dev\AmazonFreeRTOS` works, but `C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` causes build failures.

In this tutorial, the path to the AmazonFreeRTOS directory is referred to as **BASE_FOLDER**.

### Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. **Registering Your MCU Board with AWS IoT (p. 5)** is a step in the **Prerequisites (p. 4)**.

#### To configure your AWS IoT endpoint

1. Browse to the **AWS IoT** console.
2. In the navigation pane, choose **Settings**.

   Your AWS IoT endpoint is displayed in **Endpoint**. It should look like `<1234567890123>-ats.iot.<us-east-1>.amazonaws.com`. Make a note of this endpoint.

3. In the navigation pane, choose **Manage**, and then choose **Things**.

   Your device should have an AWS IoT thing name. Make a note of this name.

4. In your IDE, open `<BASE_FOLDER>\demos\common\include\aws_clientcredential.h` and specify values for the following #define constants:

   - `clientcredentialMQTT_BROKER_ENDPOINT Your AWS IoT endpoint`
   - `clientcredentialIOT_THING_NAME The AWS IoT thing name of your board`
To configure your Wi-Fi

1. Open the `aws_clientcredential.h` file.
2. Specify values for the following `#define` constants:
   - `clientcredentialWIFI_SSID` The SSID for your Wi-Fi network
   - `clientcredentialWIFI_PASSWORD` The password for your Wi-Fi network
   - `clientcredentialWIFI_SECURITY` The security type of your Wi-Fi network

   Valid security types are:
   - `eWiFiSecurityOpen` (Open, no security)
   - `eWiFiSecurityWEP` (WEP security)
   - `eWiFiSecurityWPA` (WPA security)
   - `eWiFiSecurityWPA2` (WPA2 security)

To configure your AWS IoT credentials

**Note**
To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You must format the certificate and private key for your device.

1. In a browser window, open `<BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html`.
2. Under Certificate PEM file, choose the `<ID>-certificate.pem.crt` that you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the `<ID>-private.pem.key` that you downloaded from the AWS IoT console.
4. Choose Generate and save `aws_clientcredential_keys.h`, and then save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the existing file in the directory.

**Note**
The certificate and private key are hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.

Build and Run Amazon FreeRTOS Samples

**Import the Amazon FreeRTOS Sample Code into DAVE**

1. Start DAVE.
2. In DAVE, choose File, Import. In the Import window, expand the Infineon folder, choose DAVE Project, and then choose Next.
3. In the **Import DAVE Projects** window, choose **Select Root Directory**, choose **Browse**, and then choose the XMC4800 demo project.

In the directory where you unzipped your Amazon FreeRTOS download, the demo project is located in `<BASE_FOLDER>/demos/infineon/xmc4800_iotkit/dave`.

Make sure that **Copy Projects Into Workspace** is unchecked.

4. Choose **Finish**.

The `aws_demos` project should be imported into your workspace and activated.

5. From the **Project** menu, choose **Build Active Project**.

Make sure that the project builds without errors.

---

**Run the FreeRTOS Demo**

After you have configured your project, you are ready to run the demo project on your board.

1. Use a USB cable to connect your XMC4800 IoT Connectivity Kit to your computer. The board has two microUSB connectors. Use the one labeled “X101”, where Debug appears next to it on the board’s silkscreen.

2. From the **Project** menu, choose **Rebuild Active Project** to rebuild `aws_demos` and ensure that your configuration changes are picked up.

3. Sign in to the AWS IoT console.

4. In the navigation pane, choose **Test** to open the MQTT client.
5. In Subscription topic, enter `freertos/demos/echo`, and then choose Subscribe to topic.

6. From Project Explorer, right-click `aws_demos`, choose Debug As, and then choose DAVE C/C++ Application.

7. Double-click GDB SEGGER J-Link Debugging to create a debug confirmation. Choose Debug.

8. When the debugger stops at the breakpoint in `main()`, from the Run menu, choose Resume.

In the AWS IoT console, the MQTT client from steps 4-5 should display the MQTT messages sent by your device. If you use the serial connection, you see something like this on the UART output:

0 0 [Tmr Svc] Starting key provisioning...
1 1 [Tmr Svc] Write root certificate...
2 4 [Tmr Svc] Write device private key...
3 82 [Tmr Svc] Write device certificate...
4 86 [Tmr Svc] Key provisioning done...
5 291 [Tmr Svc] Wi-Fi module initialized. Connecting to AP...
6 8046 [Tmr Svc] Wi-Fi Connected to AP. Creating tasks which use network...
7 8058 [Tmr Svc] IP Address acquired [IP Address]
8 8058 [Tmr Svc] Creating MQTT Echo Task...
9 8059 [MQTTEcho] MQTT echo attempting to connect to [MQTT Broker]. ...
10 23010 [MQTTEcho] MQTT echo connected.
11 23010 [MQTTEcho] MQTT echo test echoing task created.
12 26011 [MQTTEcho] MQTT Echo demo subscribed to freertos/demos/echo...
13 29012 [MQTTEcho] Echo successfully published 'Hello World 0'
14 32096 [Echoing] Message returned with ACK: 'Hello World 0 ACK'
15 37013 [MQTTEcho] Echo successfully published 'Hello World 1'
16 40080 [Echoing] Message returned with ACK: 'Hello World 1 ACK'
17 45014 [MQTTEcho] Echo successfully published 'Hello World 2'
18 48091 [Echoing] Message returned with ACK: 'Hello World 2 ACK'
19 53015 [MQTTEcho] Echo successfully published 'Hello World 3'
20 56087 [Echoing] Message returned with ACK: 'Hello World 3 ACK'
21 61016 [MQTTEcho] Echo successfully published 'Hello World 4'
22 64083 [Echoing] Message returned with ACK: 'Hello World 4 ACK'
23 69017 [MQTTEcho] Echo successfully published 'Hello World 5'
24 72091 [Echoing] Message returned with ACK: 'Hello World 5 ACK'
25 77018 [MQTTEcho] Echo successfully published 'Hello World 6'
26 80085 [Echoing] Message returned with ACK: 'Hello World 6 ACK'
27 85019 [MQTTEcho] Echo successfully published 'Hello World 7'
28 88086 [Echoing] Message returned with ACK: 'Hello World 7 ACK'
29 93020 [MQTTEcho] Echo successfully published 'Hello World 8'
30 96088 [Echoing] Message returned with ACK: 'Hello World 8 ACK'
31 101021 [MQTTEcho] Echo successfully published 'Hello World 9'
32 104102 [Echoing] Message returned with ACK: 'Hello World 9 ACK'
33 109022 [MQTTEcho] Echo successfully published 'Hello World 10'
34 112047 [Echoing] Message returned with ACK: 'Hello World 10 ACK'
35 117023 [MQTTEcho] Echo successfully published 'Hello World 11'
37 122068 [MQTTEcho] MQTT echo demo finished.
38 122068 [MQTTEcho] ----Demo finished----

Getting Started with the Xilinx Avnet MicroZed Industrial IoT Kit

Before you begin, see Prerequisites (p. 4).

If you do not have the Xilinx Avnet MicroZed Industrial IoT Kit, you can purchase one from Avnet.
Setting Up the MicroZed Hardware

The following diagram might be helpful when you set up the MicroZed hardware:

To set up the MicroZed board
1. Connect your computer to the USB-UART port on your MicroZed board.
2. Connect your computer to the JTAG Access port on your MicroZed board.
3. Connect a router or internet-connected Ethernet port to the Ethernet and USB-Host port on your MicroZed board.

Setting Up Your Environment

To set up Amazon FreeRTOS configurations for the MicroZed kit, you must use the Xilinx Software Development Kit (XSDK). XSDK is supported on Windows and Linux.

Download and Install XSDK

To install Xilinx software, you need a free Xilinx account.

To download the XSDK
1. Go to the Software Development Kit Standalone WebInstall Client download page.
2. Choose the option appropriate for your operating system.
3. You are directed to a Xilinx sign-in page.
   - If you have an account with Xilinx, enter your user name and password and then choose Sign in.
   - If you do not have an account, choose Create your account. After you register, you should receive an email with a link to activate your Xilinx account.
4. On the Name and Address Verification page, enter your information and then choose Next. The download should be ready to start.
5. Save the Xilinx_SDK_version_os file.

To install the XSDK

1. Open the Xilinx_SDK_version_os file.
2. In Select Edition to Install, choose Xilinx Software Development Kit (XSDK) and then choose Next.
3. On the following page of the installation wizard, under Installation Options, select Install Cable Drivers and then choose Next.
If your computer does not detect the MicroZed's USB-UART connection, install the CP210x USB-to-UART Bridge VCP drivers manually. For instructions, see the Silicon Labs CP210x USB-to-UART Installation Guide.

For more information about XSDK, see the Getting Started with Xilinx SDK on the Xilinx website.

Download and Configure Amazon FreeRTOS

After you set up your environment, you can download Amazon FreeRTOS.

Download Amazon FreeRTOS

1. Browse to the AWS IoT console.
2. In the navigation pane, choose Software.
4. Under Software Configurations, find Connect to AWS IoT- Xilinx, and then choose Download.
5. Unzip the downloaded file to the AmazonFreeRTOS folder, and make a note of the folder's path.

Note
The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, C:\Users\Username\Dev\AmazonFreeRTOS works, but C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS causes build failures.

In this tutorial, the path to the AmazonFreeRTOS directory is referred to as BASE_FOLDER.
Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. This is a step in the Prerequisites (p. 4).

To configure your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the navigation pane, choose Settings.
   
   Your AWS IoT endpoint appears in the Endpoint text box. It should look like <1234567890123>-ats.iot.<us-east-1>.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose Manage, and then choose Things. Make a note of the AWS IoT thing name for your device.
4. With your AWS IoT endpoint and your AWS IoT thing name on hand, open <BASE_FOLDER>
demos\common\include\aws_clientcredential.h in your IDE, and specify values for the following #define constants:
   
   • clientcredentialMQTT_BROKER_ENDPOINT Your AWS IoT endpoint
   • clientcredentialIOT_THING_NAME Your board’s AWS IoT thing name

To configure your AWS IoT credentials

To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device as an AWS IoT thing. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You need to format the certificate and private key for your device.

1. In a browser window, open <BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html.
2. Under Certificate PEM file, choose the <ID>-certificate.pem.crt that you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the <ID>-private.pem.key that you downloaded from the AWS IoT console.
4. Choose Generate and save aws_clientcredential_keys.h, and then save the file in <BASE_FOLDER>\demos\common\include. This overwrites the existing file in the directory.

Note
The certificate and private key should be hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.

Build and Run Amazon FreeRTOS Samples

Now that you have configured your project, you are ready to build and run the demo project on your board.

Before you run the demo project, use the MQTT client in the AWS IoT console to subscribe to the demo's MQTT topic.

To subscribe to the MQTT topic
1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test** to open the MQTT client.
3. In **Subscription topic**, enter `freertos/demos/echo`, and then choose **Subscribe to topic**.

**Open the Amazon FreeRTOS Sample Code in the XSDK IDE**

1. Launch the XSDK IDE with the workspace directory set to `<BASE_FOLDER>\demos\xilinx\microzed\xsdk`.
2. Close the welcome page. From the menu, choose **Project**, and then clear **Build Automatically**.
3. From the menu, choose **File**, and then choose **Import**.
4. On the **Select** page, expand **General**, choose **Existing Projects into Workspace**, and then choose **Next**.

5. On the **Import Projects** page, choose **Select root directory**, and then enter the root directory of your demo project. To browse for the directory, choose **Browse**.

After you specify a root directory, the projects in that directory appear on the **Import Projects** page. All available projects are selected by default.
Note
If you see a warning at the top of the Import Projects page ("Some projects cannot be imported because they already exist in the workspace.") you can ignore it.

6. With all of the projects selected, choose Finish. The XSDK IDE opens all of the projects that are required for the aws_demos project to build and run on the MicroZed board.

7. From the menu, choose Window, and then choose Preferences.

8. In the navigation pane, expand Run/Debug, choose String Substitution, and then choose New.

9. In New String Substitution Variable, for Name, enter AFR_ROOT. For Value, enter the root path of the aws_demos. Choose OK, and then choose OK to save the variable and close Preferences.
Build the Amazon FreeRTOS Project

1. In the XSDK IDE, from the menu, choose **Project**, and then choose **Clean**.
2. In **Clean**, leave the options at their default values, and then choose **OK**. XSDK cleans and builds all of the projects, and then generates `.elf` files.
Note
To build all projects without cleaning them, choose Project, and then choose Build All. To build individual projects, select the project you want to build, choose Project, and then choose Build Project.

JTAG Debugging

1. Set your MicroZed board’s boot mode jumpers to the JTAG boot mode:

2. Insert your MicroSD card into the MicroSD card slot located directly under the USB-UART port.

   Note
   Before you debug, be sure to back up any content that you have on the MicroSD card.

   Your board should look similar to the following:

3. In the XSDK IDE, right-click aws_demos, choose Debug As, and then choose 1 Launch on System Hardware (System Debugger).

4. When the debugger stops at the breakpoint in main(), from the menu, choose Run, and then choose Resume.

   Note
   The first time you run the application, a new certificate-key pair is generated. For subsequent runs, you can comment out vDevModeKeyProvisioning() in the main.c file before you rebuild the images and the BOOT.bin file. This prevents the copying of the certificates and key to storage on every run.

You can opt to boot your MicroZed board from a MicroSD card or from QSPI flash to run the Amazon FreeRTOS demo project. For instructions, see Generate the Boot Image for the Amazon FreeRTOS Project (p. 49) and Run the Amazon FreeRTOS Project (p. 49).
Generate the Boot Image for the Amazon FreeRTOS Project

1. In the XSDK IDE, right-click \aws_demos\, and then choose Create Boot Image.
2. In Create Boot Image, choose Create new BIF file.
3. Next to Output BIF file path, choose Browse, and then choose \aws_demos.bif located at <BASE_FOLDER>\demos\xilinx\microzed\xsdk\aws_demos\bootimage\aws_demos.bif.
4. Choose Add.
5. On Add new boot image partition, next to File path, choose Browse, and then choose \fsbl.elf, located at <BASE_FOLDER>\lib\third_party\mcu_vendor\xilinx\fsbl\Debug\fsbl.elf.
6. For the Partition type, choose bootloader, and then choose OK.

![Image of boot image creation process]

7. On Create Boot Image, choose Create Image. On Override Files, choose OK to overwrite the existing \aws_demos.bif and generate the \BOOT.bin file at \demos\xilinx\microzed\xsdk\aws_demos\bootimage\BOOT.bin.

Run the Amazon FreeRTOS Project

To run the Amazon FreeRTOS demo project, you can boot your MicroZed board from a MicroSD card or from QSPI flash.

As you set up your MicroZed board for running the Amazon FreeRTOS demo project, refer to the diagram in Setting Up the MicroZed Hardware (p. 41). Make sure that you have connected your MicroZed board to your computer.
Boot the Amazon FreeRTOS Project from a MicroSD Card

Format the MicroSD card that is provided with the Xilinx MicroZed Industrial IoT Kit.

1. Copy the `BOOT.bin` file to the MicroSD card.
2. Insert the card into the MicroSD card slot directly under the USB-UART port.
3. Set the MicroZed boot mode jumpers to SD boot mode:

   ![SD Card](image)

4. Press the RST button to reset the device and start booting the application. You can also unplug the USB-UART cable from the USB-UART port, and then reinsert the cable.

Boot the Amazon FreeRTOS Project from QSPI flash

1. Set your MicroZed board's boot mode jumpers to the JTAG boot mode:

   ![JTAG Boot Mode](image)

2. Verify that your computer is connected to the USB-UART and JTAG Access ports. The green Power Good LED light should be illuminated.
3. In the XSDK IDE, from the menu, choose Xilinx, and then choose Program Flash.
4. In Program Flash Memory, the hardware platform should be filled in automatically. For Connection, choose your MicroZed hardware server to connect your board with your host computer.

   **Note**
   If you are using the Xilinx Smart Lync JTAG cable, you must create a hardware server in XSDK IDE. Choose New, and then define your server.
5. In **Image File**, enter the directory path to your `BOOT.bin` image file. Choose **Browse** to browse for the file instead.

6. In **Offset**, enter `0x0`.

7. In **FSBL File**, enter the directory path to your `fsbl.elf` file. Choose **Browse** to browse for the file instead.

8. Choose **Program** to program your board.

9. After the QSPI programming is complete, remove the USB-UART cable to power off the board.

10. Set your MicroZed board’s boot mode jumpers to the QSPI boot mode:
11. Insert your card into the MicroSD card slot located directly under the USB-UART port.

   **Note**
   Be sure to back up any content that you have on the MicroSD card.

12. Press the RST button to reset the device and start booting the application. You can also unplug the USB-UART cable from the USB-UART port, and then reinsert the cable.

## Troubleshooting

### General Troubleshooting Tips

- If you encounter build errors that are related to incorrect paths, try to clean and rebuild the project, as described in Build the Amazon FreeRTOS Project (p. 47).

  **Note**
  If you are using Windows, make sure that you use forward slashes when you set the string substitution variables in the Windows XSDK IDE.

### Getting Started with the FreeRTOS Windows Simulator

Before you begin, see Prerequisites (p. 4).

Amazon FreeRTOS is released as a zip file that contains the Amazon FreeRTOS libraries and sample applications for the platform you specify. To run the samples on a Windows machine, download the libraries and samples ported to run on Windows. This set of files is referred to as the FreeRTOS simulator for Windows.

### Setting Up Your Environment

1. Install the latest version of WinPCap.
3. Make sure that you have an active hard-wired Ethernet connection.

### Download and Configure Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS.

**Download Amazon FreeRTOS**

1. In the AWS IoT console, browse to the Amazon FreeRTOS page.
2. In the navigation pane, choose **Software**.
3. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
4. Choose **Download FreeRTOS Software**.
5. In the list of software configurations, find the **Connect to AWS IoT - Windows** predefined configuration for the Windows simulator, and then choose **Download**.
6. Unzip the downloaded file to the AmazonFreeRTOS folder, and make a note of the folder's path.

**Note**
The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, C:`\Users\Username\Dev\AmazonFreeRTOS` works, but C:`\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` causes build failures.
In this tutorial, the path to the AmazonFreeRTOS directory is referred to as **BASE_FOLDER**.

## Configure Your Project

### Configure Your Network Interface

1. Run the project in Visual Studio. The program enumerates your network interfaces. Find the number for your hard-wired Ethernet interface. The output should look like this:

```
0 0 [None] FreeRTOS_IPInit
1 0 [None] vTaskStartScheduler
1. rpcap://\Device\NPF_{AD01B877-A0C1-4F33-8256-EE1F4480B70D} (Network adapter 'Intel(R) Ethernet Connection (4) I219-LM' on local host)
2. rpcap://\Device\NPF_{337F7AF9-2520-4667-8EFF-2B575A98B580} (Network adapter 'Microsoft' on local host)
```

The interface that will be opened is set by "configNETWORK_INTERFACE_TO_USE" which should be defined in FreeRTOSConfig.h Attempting to open interface number 1.

You might see output in the debugger that says **Cannot find or open the PDB file**. You can ignore these messages.

After you have identified the number for your hard-wired Ethernet interface, close the application window.

2. Open `<BASE_FOLDER>\demos\pc\windows\common\config_files\FreeRTOSConfig.h` and set `configNETWORK_INTERFACE_TO_USE` to the number that corresponds to your hard-wired network interface.

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. This is a step in the Prerequisites (p. 4).

### To configure your AWS IoT endpoint

1. Browse to the **AWS IoT console**.
2. In the navigation pane, choose **Settings**.

Your AWS IoT endpoint appears in the **Endpoint** text box. It should look like `<1234567890123>-ats.iot.<us-east-1>.amazonaws.com`. Make a note of this endpoint.
3. In the navigation pane, choose Manage, and then choose Things. Make a note of the AWS IoT thing name for your device.

4. With your AWS IoT endpoint and your AWS IoT thing name on hand, open `<BASE_FOLDER>\demos\common\include\aws_clientcredential.h` in your IDE, and specify values for the following #define constants:
   - `clientcredentialMQTT_BROKER_ENDPOINT` Your AWS IoT endpoint
   - `clientcredentialIOT_THING_NAME` Your board’s AWS IoT thing name

To configure your AWS IoT credentials

To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device as an AWS IoT thing. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You need to format the certificate and private key for your device.

1. In a browser window, open `<BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html`.
2. Under Certificate PEM file, choose the `<ID>-certificate.pem.crt` that you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the `<ID>-private.pem.key` that you downloaded from the AWS IoT console.
4. Choose Generate and save aws_clientcredential_keys.h, and then save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the existing file in the directory.

   **Note**

   The certificate and private key should be hard-coded for demonstration purposes only. Production-level applications should store these files in a secure location.

Build and Run Amazon FreeRTOS Samples

Load the Amazon FreeRTOS Sample Code into Visual Studio

1. In Visual Studio, from the File menu, choose Open. Choose File/Solution, navigate to `<BASE_FOLDER>\demos\pc\windows\visual_studio\aws_demos.sln`, and then choose Open.
2. From the Build menu, choose Build Solution, and make sure the solution builds without errors or warnings.

Run the Amazon FreeRTOS Samples

1. Rebuild your Visual Studio project to pick up changes made in the header files.
2. Sign in to the AWS IoT console.
3. In the navigation pane, choose Test to open the MQTT client.
4. In Subscription topic, enter `freertos/demos/echo`, and then choose Subscribe to topic.
5. From the Debug menu in Visual Studio, choose Start Debugging.

   In the AWS IoT console, the MQTT client displays the messages received from the FreeRTOS Windows simulator.
Getting Started with the Nordic nRF52840-DK

Amazon FreeRTOS support for the Nordic nRF52840-DK is in public beta release. BLE demos are subject to change.

Before you begin, see Prerequisites (p. 4).

If you do not have the Nordic nRF52840-DK, you can purchase one from Nordic.

Setting Up the Nordic Hardware

Connect your host computer to the USB port labeled J2, located directly above the coin cell battery holder on your Nordic nRF52840 board.

For more information about setting up the Nordic nRF52840-DK, see the nRF52840 Development Kit User Guide.

Setting Up Your Environment

Download and Install Segger Embedded Studio

Amazon FreeRTOS supports Segger Embedded Studio as a development environment for the Nordic nRF52840-DK.

To set up your environment, you need to download and install Segger Embedded Studio.

1. Go to the Segger Embedded Studio Downloads page and choose the Embedded Studio for ARM option for your operating system.
2. Run the installer and follow the prompts to completion.

Establish a Serial Connection

After you connect your computer to your Nordic nRF52840 board and install Segger Embedded Studio, open a terminal tool, like PuTTY, Tera Term, or GNU Screen. Configure the terminal to connect to your board by a serial connection. Set the COM port to JLink CDC UART Port with the following serial port settings:

- Baud Rate: 115200
- Data: 8 bit
- Parity: None
- Stop: 1 bit
- Flow Control: None

Note
Depending on your terminal tool, the serial port settings might vary in name.

Download and Configure Amazon FreeRTOS

After you set up your hardware and environment, you can download Amazon FreeRTOS.
Download Amazon FreeRTOS

To download Amazon FreeRTOS for the Nordic nRF52840-DK, go to the Amazon FreeRTOS GitHub page and clone the repository. The Amazon FreeRTOS BLE library is still in public beta, so you need to switch branches to access the code for the Nordic nRF52840-DK board. Check out the branch named `feature/ble-beta`.

**Note**

The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS download is 122 characters. To accommodate the files in the Amazon FreeRTOS projects, make sure that the path to the AmazonFreeRTOS directory is fewer than 98 characters long. For example, `C:\Users\Username\Dev\AmazonFreeRTOS` works, but `C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` causes build failures.

In this tutorial, the path to the AmazonFreeRTOS directory is referred to as `BASE_FOLDER`.

Configure Your Project

To run the demo, you must configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your board must be registered as an AWS IoT thing. This is a step in the Prerequisites (p. 4).

**To configure your AWS IoT endpoint**

1. Browse to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
   
   Your AWS IoT endpoint appears in the **Endpoint** text box. It should look like `{<1234567890123>-ats.iot.<us-east-1>.amazonaws.com}. Make a note of this endpoint.
3. In the navigation pane, choose **Manage**, and then choose **Things**. Make a note of the AWS IoT thing name for your device.
4. With your AWS IoT endpoint and your AWS IoT thing name on hand, open `<BASE_FOLDER>\demos\common\include\aws_clientcredential.h` in your IDE, and specify values for the following `#define` constants:
   - `clientcredentialMQTT_BROKER_ENDPOINT` *Your AWS IoT endpoint*
   - `clientcredentialIOT_THING_NAME` *Your board’s AWS IoT thing name*

**To configure your AWS IoT credentials**

To configure your AWS IoT credentials, you need the private key and certificate that you downloaded from the AWS IoT console when you registered your device as an AWS IoT thing. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project. You need to format the certificate and private key for your device.

1. In a browser window, open `<BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the `{<ID>-certificate.pem.crt}` that you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the `{<ID>-private.pem.key}` that you downloaded from the AWS IoT console.
4. Choose **Generate and save aws_clientcredential_keys.h**, and then save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the existing file in the directory.
To enable the demo

1. Check that the BLE GATT Demo is enabled. Go to `<BASE_FOLDER>/demos/nordic/nrf52840-dk/common/config_files/aws_ble_config.h`, and make sure that `bleconfigENABLE_GATT_DEMO` is set to 1.

2. Open `<BASE_FOLDER>/demos/common/demo_runner/aws_demo_runner.c`, and in the demo declarations, uncomment `extern void vStartMQTTBLEEchoDemo( void );`. In the `DEMO_RUNNER_RunDemos` definition, uncomment `vStartMQTTBLEEchoDemo();`.

Build and Run Amazon FreeRTOS Samples

After you download Amazon FreeRTOS and configure your demo project, you are ready to build and run the demo project on your board.

Open Segger Embedded Studio. From the top menu, choose File, choose Open Solution, and then navigate to the project file `<BASE_FOLDER>/demos/nordic/nrf52840-dk/ses/aws_demos_ble.emProject`

From the top menu, choose View, and then choose Debug Terminal to display information from your serial connection terminal.

To build the BLE demo, right-click the `aws_ble_demos` demo project, and choose Build.

Note
If this is your first time using Segger Embedded Studio, you might see you a warning "No license for commercial use". Segger Embedded Studio can be used free of charge for Nordic Semiconductor devices. Choose Activate Your Free License, and follow the instructions.

To run the BLE demo on your board, from the Segger Embedded Studio menu, choose Debug, and then choose Go.

For more information about completing the demo with the Amazon FreeRTOS BLE Mobile SDK demo application as the mobile MQTT proxy, see MQTT over BLE Demo Application.
Amazon FreeRTOS Developer Guide

This section contains information required for writing embedded applications with Amazon FreeRTOS.

Topics

• Amazon FreeRTOS Architecture (p. 58)
• FreeRTOS Kernel Fundamentals (p. 58)
• Amazon FreeRTOS Libraries (p. 64)
• Amazon FreeRTOS Over-the-Air Updates (p. 108)
• Amazon FreeRTOS Console User Guide (p. 153)

Amazon FreeRTOS Architecture

Amazon FreeRTOS is intended for use on embedded microcontrollers. It is typically flashed to devices as a single compiled image with all of the components required for the device application. This image combines functionality for the application written by the embedded developer, software libraries provided by Amazon, the FreeRTOS kernel, and drivers and board support packages (BSPs) for the hardware platform. Independent of the individual microcontroller being used, embedded application developers can expect the same standardized interfaces to the FreeRTOS kernel and all Amazon FreeRTOS software libraries.

FreeRTOS Kernel Fundamentals

The FreeRTOS kernel is a real-time operating system that supports numerous architectures. It is ideal for building embedded microcontroller applications. It provides:

• A multitasking scheduler.
• Multiple memory allocation options (including the ability to create completely statically allocated systems).
• Intertask coordination primitives, including task notifications, message queues, multiple types of semaphore, and stream and message buffers.

The FreeRTOS kernel never performs non-deterministic operations, such as walking a linked list, inside a critical section or interrupt. The FreeRTOS kernel includes an efficient software timer implementation that does not use any CPU time unless a timer needs servicing. Blocked tasks do not require time-consuming periodic servicing. Direct-to-task notifications allow fast task signaling, with practically no RAM overhead. They can be used in the majority of intertask and interrupt-to-task signaling scenarios.

The FreeRTOS kernel is designed to be small, simple, and easy to use. A typical RTOS kernel binary image is in the range of 4000 to 9000 bytes.

FreeRTOS Kernel Scheduler

An embedded application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context, with no dependency on other tasks. Only one task in the application is running at any point in time. The real-time RTOS scheduler determines when each task should run. Each task is provided with its own stack. When a task is swapped out so another task can run, the task's execution context is saved to the task stack so it can be restored when the same task is later swapped back in to resume its execution.

To provide deterministic real-time behavior, the FreeRTOS tasks scheduler allows tasks to be assigned strict priorities. RTOS ensures the highest priority task that is able to execute is given processing time. This requires sharing processing time between tasks of equal priority if they are ready to run simultaneously. FreeRTOS also creates an idle task that executes only when no other tasks are ready to run.

Memory Management

This section provides information about kernel memory allocation and application memory management.

Kernel Memory Allocation

The RTOS kernel needs RAM each time a task, queue, or other RTOS object is created. The RAM can be allocated:

• Statically at compile time.
• Dynamically from the RTOS heap by the RTOS API object creation functions.

When RTOS objects are created dynamically, using the standard C library malloc() and free() functions is not always appropriate for a number of reasons:

• They might not be available on embedded systems.
• They take up valuable code space.
• They are not typically thread-safe.
• They are not deterministic.

For these reasons, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, so you can provide an application-specific implementation appropriate for the real-time system you're developing. When the
RTOS kernel requires RAM, it calls `pvPortMalloc()` instead of `malloc()`(). When RAM is being freed, the RTOS kernel calls `vPortFree()` instead of `free()`.

**Application Memory Management**

When applications need memory, they can allocate it from the FreeRTOS heap. FreeRTOS offers several heap management schemes that range in complexity and features. You can also provide your own heap implementation.

The FreeRTOS kernel includes five heap implementations:

- **heap_1**
  
  Is the simplest implementation. Does not permit memory to be freed.

- **heap_2**
  
  Permits memory to be freed, but not does coalescence adjacent free blocks.

- **heap_3**
  
  Wraps the standard `malloc()` and `free()` for thread safety.

- **heap_4**
  
  Coalesces adjacent free blocks to avoid fragmentation. Includes an absolute address placement option.

- **heap_5**
  
  Is similar to heap_4. Can span the heap across multiple, non-adjacent memory areas.

**Intertask Coordination**

This section contains information about FreeRTOS primitives.

**Queues**

Queues are the primary form of intertask communication. They can be used to send messages between tasks and between interrupts and tasks. In most cases, they are used as thread-safe First In First Out (FIFO) buffers with new data being sent to the back of the queue. (Data can also be sent to the front of the queue.) Messages are sent through queues by copy, meaning the data (which can be a pointer to larger buffers) is itself copied into the queue rather than simply storing a reference to the data.

Queue APIs permit a block time to be specified. When a task attempts to read from an empty queue, the task is placed into the Blocked state until data becomes available on the queue or the block time elapses. Tasks in the Blocked state do not consume any CPU time, allowing other tasks to run. Similarly, when a task attempts to write to a full queue, the task is placed into the Blocked state until space becomes available in the queue or the block time elapses. If more than one task blocks on the same queue, the task with the highest priority is unblocked first.

Other FreeRTOS primitives, such as direct-to-task notifications and stream and message buffers, offer lightweight alternatives to queues in many common design scenarios.

**Semaphores and Mutexes**

The FreeRTOS kernel provides binary semaphores, counting semaphores, and mutexes for both mutual exclusion and synchronization purposes.
Binary semaphores can only have two values. They are a good choice for implementing synchronization (either between tasks or between tasks and an interrupt). Counting semaphores take more than two values. They allow many tasks to share resources or perform more complex synchronization operations.

Mutexes are binary semaphores that include a priority inheritance mechanism. This means that if a high priority task blocks while attempting to obtain a mutex that is currently held by a lower priority task, the priority of the task holding the token is temporarily raised to that of the blocking task. This mechanism is designed to ensure the higher priority task is kept in the Blocked state for the shortest time possible, to minimize the priority inversion that has occurred.

Direct-to-Task Notifications

Task notifications allow tasks to interact with other tasks, and to synchronize with interrupt service routines (ISRs), without the need for a separate communication object like a semaphore. Each RTOS task has a 32-bit notification value that is used to store the content of the notification, if any. An RTOS task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value.

RTOS task notifications can be used as a faster and lightweight alternative to binary and counting semaphores and, in some cases, queues. Task notifications have both speed and RAM footprint advantages over other FreeRTOS features that can be used to perform equivalent functionality. However, task notifications can only be used when there is only one task that can be the recipient of the event.

Stream Buffers

Stream buffers allow a stream of bytes to be passed from an interrupt service routine to a task, or from one task to another. A byte stream can be of arbitrary length and does not necessarily have a beginning or an end. Any number of bytes can be written at one time, and any number of bytes can be read at one time. Stream buffer functionality is enabled by including the `<BASE_DIR>/libs/FreeRTOS/stream_buffer.c` source file in your project.

Stream buffers assume there is only one task or interrupt that writes to the buffer (the writer), and only one task or interrupt that reads from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupt service routines, but it is not safe to have multiple writers or readers.

The stream buffer implementation uses direct to task notifications. Therefore, calling a stream buffer API that places the calling task into the Blocked state can change the calling task's notification state and value.

Sending Data

`xStreamBufferSend()` is used to send data to a stream buffer in a task. `xStreamBufferSendFromISR()` is used to send data to a stream buffer in an interrupt service routine (ISR).

`xStreamBufferSend()` allows a block time to be specified. If `xStreamBufferSend()` is called with a non-zero block time to write to a stream buffer and the buffer is full, the task is placed into the Blocked state until space becomes available or the block time expires.

`sbSEND_COMPLETED()` and `sbSEND_COMPLETED_FROM_ISR()` are macros that are called (internally by the FreeRTOS API) when data is written to a stream buffer. It takes the handle of the stream buffer that was updated. Both of these macros check to see if there is a task blocked on the stream buffer waiting for data, and if so, removes the task from the Blocked state.

You can change this default behavior by providing your own implementation of `sbSEND_COMPLETED()` in `FreeRTOSConfig.h`. This is useful when a stream buffer is used to pass data between cores on a multicore processor. In that scenario, `sbSEND_COMPLETED()` can be implemented to
generate an interrupt in the other CPU core, and the interrupt's service routine can then use the
\texttt{xStreamBufferSendCompletedFromISR()} API to check, and if necessary unblock, a task that is
waiting for the data.

\textbf{Receiving Data}

\texttt{xStreamBufferReceive()} is used to read data from a stream buffer in a task.
\texttt{xStreamBufferReceiveFromISR()} is used to read data from a stream buffer in an interrupt service
routine (ISR).

\texttt{xStreamBufferReceive()} allows a block time to be specified. If \texttt{xStreamBufferReceive()} is called
with a non-zero block time to read from a stream buffer and the buffer is empty, the task is placed into
the Blocked state until either a specified amount of data becomes available in the stream buffer, or the
block time expires.

The amount of data that must be in the stream buffer before a task is unblocked is called the
stream buffer's trigger level. A task blocked with a trigger level of 10 is unblocked when at least 10
bytes are written to the buffer or the task's block time expires. If a reading task's block time expires
before the trigger level is reached, the task receives any data written to the buffer. The trigger
level of a task must be set to a value between 1 and the size of the stream buffer. The trigger
level of a stream buffer is set when \texttt{xStreamBufferCreate()} is called. It can be changed by calling
\texttt{xStreamBufferSetTriggerLevel()}.

\texttt{sbRECEIVE_COMPLETED()} and \texttt{sbRECEIVE_COMPLETED_FROM_ISR()} are macros that are called
(internally by the FreeRTOS API) when data is read from a stream buffer. The macros check to see
if there is a task blocked on the stream buffer waiting for space to become available within the
buffer, and if so, removes the task from the Blocked state. You can change the default behavior of
\texttt{sbRECEIVE_COMPLETED()} by providing an alternative implementation in \texttt{FreeRTOSConfig.h}.

\textbf{Message Buffers}

Message buffers allow variable length discrete messages to be passed from an interrupt service
routine to a task, or from one task to another. For example, messages of length 10, 20 and 123
bytes can all be written to, and read from, the same message buffer. A 10-byte message can only
be read as a 10-byte message, not as individual bytes. Message buffers are built on top of stream
buffer implementation. Message buffer functionality is enabled by including the \texttt{<BASE_DIR>/libs/
FreeRTOS/stream_buffer.c} source file in your project.

Message buffers assume there is only one task or interrupt that writes to the buffer (the writer), and only
one task or interrupt that reads from the buffer (the reader). It is safe for the writer and reader to be
different tasks or interrupt service routines, but it is not safe to have multiple writers or readers.

The message buffer implementation uses direct to task notifications. Therefore, calling a stream buffer
API that places the calling task into the Blocked state can change the calling task's notification state and
value.

To enable message buffers to handle variable-sized messages, the length of each message is written into
the message buffer before the message itself. The length is stored in a variable of type \texttt{size_t}, which is
typically 4 bytes on a 32-byte architecture. Therefore, writing a 10-byte message into a message buffer
actually consumes 14 bytes of buffer space. Likewise, writing a 100-byte message into a message buffer
actually uses 104 bytes of buffer space.

\textbf{Sending Data}

\texttt{xMessageBufferSend()} is used to send data to a message buffer from a task.
\texttt{xMessageBufferSendFromISR()} is used to send data to a message buffer from an interrupt service
routine (ISR).
Software Timers

xMessageBufferSend() allows a block time to be specified. If xMessageBufferSend() is called with a non-zero block time to write to a message buffer and the buffer is full, the task is placed into the Blocked state until either space becomes available in the message buffer, or the block time expires.

sbSEND_COMPLETED() and sbSEND_COMPLETED_FROM_ISR() are macros that are called (internally by the FreeRTOS API) when data is written to a stream buffer. It takes a single parameter, which is the handle of the stream buffer that was updated. Both of these macros check to see if there is a task blocked on the stream buffer waiting for data, and if so, they remove the task from the Blocked state.

You can change this default behavior by providing your own implementation of sbSEND_COMPLETED() in FreeRTOSConfig.h. This is useful when a stream buffer is used to pass data between cores on a multicore processor. In that scenario, sbSEND_COMPLETED() can be implemented to generate an interrupt in the other CPU core, and the interrupt's service routine can then use the xStreamBufferSendCompletedFromISR() API to check, and if necessary unblock, a task that was waiting for the data.

Receiving Data

xMessageBufferReceive() is used to read data from a message buffer in a task.
xMessageBufferReceiveFromISR() is used to read data from a message buffer in an interrupt service routine (ISR). xMessageBufferReceive() allows a block time to be specified. If xMessageBufferReceive() is called with a non-zero block time to read from a message buffer and the buffer is empty, the task is placed into the Blocked state until either data becomes available, or the block time expires.

sbRECEIVE_COMPLETED() and sbRECEIVE_COMPLETED_FROM_ISR() are macros that are called (internally by the FreeRTOS API) when data is read from a stream buffer. The macros check to see if there is a task blocked on the stream buffer waiting for space to become available within the buffer, and if so, removes the task from the Blocked state. You can change the default behavior of sbRECEIVE_COMPLETED() by providing an alternative implementation in FreeRTOSConfig.h.

Software Timers

A software timer allows a function to be executed at a set time in the future. The function executed by the timer is called the timer's callback function. The time between a timer being started and its callback function being executed is called the timer's period. The FreeRTOS kernel provides an efficient software timer implementation because:

- It does not execute timer callback functions from an interrupt context.
- It does not consume any processing time unless a timer has actually expired.
- It does not add any processing overhead to the tick interrupt.
- It does not walk any link list structures while interrupts are disabled.

Low Power Support

Like most embedded operating systems, the FreeRTOS kernel uses a hardware timer to generate periodic tick interrupts, which are used to measure time. The power saving of regular hardware timer implementations is limited by the necessity to periodically exit and then re-enter the low power state to process tick interrupts. If the frequency of the tick interrupt is too high, the energy and time consumed entering and exiting a low power state for every tick outweighs any potential power saving gains for all but the lightest power saving modes.

To address this limitation, FreeRTOS includes a tickless timer mode for low-power applications. The FreeRTOS tickless idle mode stops the periodic tick interrupt during idle periods (periods when there are no application tasks that are able to execute), and then makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted. Stopping the tick interrupt allows the microcontroller to
remain in a deep power saving state until either an interrupt occurs, or it is time for the RTOS kernel to transition a task into the ready state.

Amazon FreeRTOS Libraries

Amazon FreeRTOS libraries provide additional functionality to the FreeRTOS kernel and its internal libraries. You can use Amazon FreeRTOS libraries for networking and security in embedded applications. Amazon FreeRTOS libraries also enable your applications to interact with AWS IoT services.

You can download versions of Amazon FreeRTOS that are configured for Amazon FreeRTOS-qualified platforms from the Amazon FreeRTOS console. For a list of qualified platforms, see the Amazon FreeRTOS Partners website. Amazon FreeRTOS is also available on GitHub.

Amazon FreeRTOS Porting Libraries

The following porting libraries are included in configurations of Amazon FreeRTOS that are available for download on the Amazon FreeRTOS console. These libraries are platform-dependent. Their contents change according to your hardware platform.

Amazon FreeRTOS Porting Libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth Low Energy (BLE)</td>
<td>Using the Amazon FreeRTOS Bluetooth Low Energy library, your microcontroller can communicate with the AWS IoT MQTT broker through a gateway device. For more information, see Amazon FreeRTOS Bluetooth Low Energy</td>
</tr>
</tbody>
</table>
### Library Reference

**Library**
(Beta) (p. 72).

**Note**
The Amazon FreeRTOS BLE library is in public beta.

**Other Amazon FreeRTOS Reference**

IOT Over-the-Air (OTA) Agent library connects your Amazon FreeRTOS device to the AWS IoT OTA agent. For more information, see Amazon FreeRTOS Over-the-Air (OTA) Agent Library (p. 93).
You can use the FreeRTOS + POSIX library to port POSIX-compliant applications to the Amazon FreeRTOS ecosystem.

For more information, see FreeRTOS + POSIX.

For more information, see Amazon FreeRTOS Secure Sockets Library (p. 97).
<table>
<thead>
<tr>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FreeRTOS +TCP API Reference</strong></td>
<td>FreeRTOS +TCP is a scalable, open source and thread safe TCP/IP stack for FreeRTOS. For more information, see FreeRTOS +TCP.</td>
</tr>
<tr>
<td><strong>Amazon FreeRTOS Wi-Fi Library</strong></td>
<td>The Amazon FreeRTOS Wi-Fi library enables you to interface with your microcontroller's lower-level wireless stack. For more information, see Amazon FreeRTOS Wi-Fi Library (p. 103).</td>
</tr>
</tbody>
</table>
The Amazon FreeRTOS PKCS #11 library is a reference implementation of the Public Key Cryptography Standard #11, to support provisioning and TLS client authentication.

For more information, see Amazon FreeRTOS Public Key Cryptography Standard (PKCS) #11 Library (p. 95).

For more information, see Amazon FreeRTOS Transport Layer Security (TLS) (p. 103).
Amazon FreeRTOS Application Libraries

You can optionally include the following standalone application libraries in your Amazon FreeRTOS configuration to interact with AWS IoT.

Amazon FreeRTOS application libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>API</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass</td>
<td>Greengrass API</td>
<td>Reference (Legacy)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQTT</td>
<td>MQTT</td>
<td>MQTT API Reference (Beta)</td>
</tr>
</tbody>
</table>

The Amazon FreeRTOS Greengrass library connects your Amazon FreeRTOS device to AWS IoT Greengrass. For more information, see Amazon FreeRTOS AWS IoT Greengrass Discovery Library (p. 85).

The Amazon FreeRTOS MQTT library provides a MQTT client API for your Amazon FreeRTOS device to publish and subscribe to.
MQTT topics. MQTT is the protocol that devices use to interact with AWS IoT.

For more information about the legacy Amazon FreeRTOS MQTT library, see Amazon FreeRTOS MQTT Library (Legacy) (p. 90).

For more information about the new Amazon FreeRTOS MQTT library, in public beta, see Amazon FreeRTOS MQTT Library (Beta) (p. 87).
The AWS IoT Device Shadow library enables your Amazon FreeRTOS device to interact with AWS IoT device shadows. For more information, see Amazon FreeRTOS AWS IoT Device Shadow Library (p. 101).
Amazon FreeRTOS Bluetooth Low Energy Library (Beta)

Overview

The Bluetooth Low Energy (BLE) Library is in public beta release for Amazon FreeRTOS and is subject to change.

Amazon FreeRTOS supports publishing and subscribing to MQTT topics over Bluetooth Low Energy (BLE) through a proxy device, such as a mobile phone. With the Amazon FreeRTOS BLE library, your microcontroller can securely communicate with the AWS IoT MQTT broker.
Using the Amazon FreeRTOS BLE Mobile SDKs, you can write native mobile applications that communicate with the embedded applications on your microcontroller over BLE. For more information about the Amazon FreeRTOS BLE Mobile SDKs, see Mobile SDKs for Amazon FreeRTOS Bluetooth Devices (p. 81). Amazon FreeRTOS BLE uses Amazon Cognito for user authentication on mobile devices.

In addition to supporting MQTT, the Amazon FreeRTOS BLE library includes services for configuring Wi-Fi networks. The Amazon FreeRTOS BLE library also includes some middleware and lower-level APIs for more direct control over your BLE stack. The source files for the Amazon FreeRTOS BLE library are located in AmazonFreeRTOS/lib/bluetooth_low_energy.

**Amazon FreeRTOS BLE Architecture**

The Amazon FreeRTOS BLE library is made up of three layers: services, middleware, and low-level wrappers.

**Services**

The Amazon FreeRTOS BLE services layer consists of three Generic Attributes (GATT) services that leverage the middleware APIs: Device Information, Wi-Fi Provisioning, and MQTT Communications over BLE. For more information, see Services (p. 74).

**Middleware**

Amazon FreeRTOS BLE middleware is an abstraction from the lower-level APIs. The middleware APIs make up a more user-friendly interface to the BLE stack. For more information, see Middleware (p. 74).

**Low-level Wrappers**

The low-level Amazon FreeRTOS BLE wrappers are an abstraction from the manufacturer's BLE stack. Low-level wrappers offer a common set of APIs for direct control over the hardware. The low-level APIs
optimize RAM usage, but are limited in functionality. To use the Amazon FreeRTOS BLE services, you interact with the BLE service APIs, which demand more resources than the low-level APIs.

**Dependencies and Requirements**

Only the MQTT over BLE and Wi-Fi Provisioning services have library dependencies.

<table>
<thead>
<tr>
<th>GATT Service</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQTT over BLE</td>
<td>Amazon FreeRTOS MQTT Library (Beta) (p. 87)</td>
</tr>
<tr>
<td>Wi-Fi Provisioning</td>
<td>Amazon FreeRTOS Wi-Fi Library (p. 103)</td>
</tr>
</tbody>
</table>

To communicate with the AWS IoT MQTT broker, you must have an AWS account and you must register your devices as AWS IoT things. For more information about setting up, see the AWS IoT Developer Guide.

Amazon FreeRTOS BLE uses Amazon Cognito for user authentication on your mobile device. To use MQTT proxy services, you must create an Amazon Cognito identity and user pools. Each Amazon Cognito Identity must have the appropriate policy attached to it. For more information, see the Amazon Cognito Developer Guide.

**Features**

**Services**

**Device Information**

The Device Information service gathers information about your microcontroller, including:

- The version of Amazon FreeRTOS that your device is using.
- The AWS IoT endpoint of the account for which the device is registered.
- BLE Maximum Transmission Unit (MTU).

**Wi-Fi Provisioning**

The Wi-Fi Provisioning service enables microcontrollers with Wi-Fi capabilities to do the following:

- List networks in range.
- Save networks and network credentials to flash memory.
- Set network priority.
- Delete networks and network credentials from flash memory.

**MQTT over BLE**

The MQTT over BLE service connects your microcontroller to Bluetooth-enabled mobile devices to indirectly connect to the AWS IoT cloud with AWS Mobile SDKs. The microcontroller functions as an MQTT client, the mobile device as an MQTT proxy, and the AWS IoT cloud as the MQTT server.

**Middleware**

Using middleware APIs, you can register several callbacks, across multiple layers, to a single event.
Flexible Callback Subscription

Suppose your BLE hardware disconnects, and the MQTT over BLE service needs to detect the disconnection. An application that you wrote might also need to detect the same disconnection event. The BLE middleware can route the event to different parts of the code where you have registered callbacks, without making the higher layers compete for lower-level resources.

Source and Header Files

The following tree diagram shows the required source and header files, along with their location in the Amazon FreeRTOS directory structure. The project must also build the source files of the dependent libraries.

```plaintext
Amazon FreeRTOS
  |  + - lib
  |  |  + - bluetooth_low_energy
  |  |  |  + - aws_ble_event_manager.c
  |  |  |  + - aws_ble_gap.c [Middleware GAP]
  |  |  |  + - aws_ble_gatt.c [Middleware GATT]
  |  |  + - portable [Wrappers, wrapping APIs in lib/include/bluetooth_low_energy]
  |  |  + - services
  |  |  |  + - device_information [Service providing device info to the phone APP]
  |  |  |  + - mqtt_ble [Used to do MQTT over BLE]
  |  |  |  + - wifi_provisioning [WIFI provisioning service over BLE]
  |  |  + - include
  |  |  |  + - bluetooth_low_energy [Wrapping APIs in lib/include/bluetooth_low_energy]
  |  |  |  + - bt_hal_avsrc_profile.h
  |  |  |  + - bt_hal_gatt_client.h
  |  |  |  + - bt_hal_gatt_server.h
  |  |  |  + - bt_hal_gatt_types.h
  |  |  |  + - bt_hal_manager_adapter_ble.h
  |  |  |  + - bt_hal_manager_adapter_classic.h
  |  |  |  + - bt_hal_manager.h
  |  |  |  + - bt_hal_manager_types.h
  |  |  + - private [For internal library use only!]
  |  |  |  + - aws_ble_internals.h
  |  |  |  + - aws_ble_config_defaults.h
  |  |  |  + - aws_ble_event_manager.h
  |  |  + - aws_ble.h
  |  + - aws_ble_device_information.h
  + - aws_ble_services_init.h
  + - aws_ble_wifi_provisioning.h
```

Amazon FreeRTOS BLE Library Configuration File

Applications that use the Amazon FreeRTOS MQTT over BLE service must provide an `aws_ble_config.h` header file, in which configuration parameters are defined. Undefined configuration parameters take the default values specified in `lib/include/private/aws_ble_config_defaults.h`.

Optimization

When optimizing your board's performance, consider the following:

- Low-level APIs use less RAM, but offer limited functionality.
• You can set the `bleconfigMAX_NETWORK` parameter in the `aws_ble_config.h` header file to a lower value to reduce the amount of stack consumed.

• You can delete unused services to save RAM.

• You can increase the MTU size to its maximum value to limit message buffering, and make code run faster and consume less RAM.

### Usage Restrictions

By default, the Amazon FreeRTOS BLE library sets the `eBTpropertySecureConnectionOnly` property to `TRUE`, which places the device in a Secure Connections Only mode. As specified by the Bluetooth Core Specification v5.0, Vol 3, Part C, 10.2.4, when a device is in a Secure Connections Only mode, the highest LE security mode 1 level, level 4, is required for access to any attribute that has permissions higher than the lowest LE security mode 1 level, level 1. At the LE security mode 1 level 4, a device must have input and output capabilities for numeric comparison.

To use a lower LE security level, set `eBTpropertySecureConnectionOnly` to `FALSE`, by calling the API `pxSetDeviceProperty` with the property `eBTpropertySecureConnectionOnly`.

For information about LE security modes, see the Bluetooth Core Specification v5.0, Vol 3, Part C, 10.2.1.

### Initialization

If your application interacts with the BLE stack through middleware, you only need to initialize the middleware.

**Middleware**

Middleware takes care of initializing the lower layers of the stack.

**To initialize the middleware**

1. You must initialize any BLE hardware drivers before you call the BLE middleware API.

2. Enable BLE.

   ```c
   const BTInterface_t * pxIface = BTGetBluetoothInterface();
   xStatus = pxIface->pxEnable( 0 );
   ```

3. To initialize BLE, call `BLE_Init`, along with a set of desired properties, such as secure connection mode, device name, and MTU size.

   ```c
   xStatus = BLE_Init( &xServerUUID, xDeviceProperties, MAX_PROPERTIES );
   ```

### Low-level APIs

If you don't want to use the Amazon FreeRTOS BLE GATT services, you can bypass the middleware and interact directly with the low-level APIs to save resources.

**To initialize the low-level APIs**

1. Driver initialization is not part of the BLE low-level APIs. You must initialize any BLE hardware drivers before you call the APIs.
2. The BLE low-level API provides an enable/disable call to the BLE stack for optimizing power and resources. Before calling the APIs, you must enable BLE.

```c
const BTInterface_t * pxIface = BTGetBluetoothInterface();
xStatus = pxIface->pxEnable( 0 );
```

3. The Bluetooth manager contains APIs that are common to both BLE and Bluetooth classic. The callbacks for the common manager must be initialized second.

```c
xStatus = xBTInterface.pxBTInterface->pxBtManagerInit( &xBTManagerCb );
```

4. The BLE adapter fits on top of the common API. You must initialize its callbacks like you initialized the common API.

```c
xBTInterface.pxBTLeAdapterInterface = ( BTBleAdapter_t * ) xBTInterface.pxBTInterface->pxGetLeAdapter();
xStatus = xBTInterface.pxBTLeAdapterInterface->pxBleAdapterInit( &xBTBleAdapterCb );
```

5. Register your new user application.

```c
xBTInterface.pxBTLeAdapterInterface->pxRegisterBleApp( pxAppUuid );
```

6. Initialize the callbacks to the GATT servers.

```c
xBTInterface.pxBTLeAdapterInterface->pxGetGattServerInterface();
xStatus = xBTInterface.pxBTLeAdapterInterface->pxBleAdapterInit( &xBTBleAdapterCb );
```

After you initialize the BLE adapter, you can add a GATT server. You can register only one GATT server at a time.

```c
xBTInterface.pxBTLeAdapterInterface->pxRegisterServer( pxAppUuid );
```

7. Set application properties like secure connection only and MTU size.

```c
xBTInterface.pxBTInterface->pxSetDeviceProperty( &pxProperty[ usIndex ] );
```

---

**API Reference**

For a full API reference, see Bluetooth Low Energy (BLE) API Reference.

**Example Usage**

**Advertising**

1. Set advertisement parameters.

```c
BLEAdvertisementParams_t xAdvParams =
{
    .bIncludeTxPower = true,
}
```
2. Start advertisement.

```c
void vSetAdvCallback ( BTStatus_t xStatus )
{
  if( xStatus == eBTStatusSuccess )
  {
    ( void ) BLE_StartAdv( vStartAdvCallback );
  }
}
```

## Adding a New Service

1. Allocate memory for new service.

```c
xStatus = BLE_CreateService( &pxGattDemoService, gattDemoNUM_CHARS, gattDemoNUM_CHAR_DESCRS, xNumDescrsPerChar, gattDemoNUM_INCLUDED_SERVICES );
```

2. Create the service.

```c
pxGattDemoService->xAttributeData.xUuid = xServiceUUID;
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xAttributeData.xUuid = xClientCharCfgUUID;
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xAttributeData.pucData = NULL;
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xAttributeData.xSize = 0;
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xPermissions =
( eBTPermReadEncryptedMitm | eBTPermWriteEncryptedMitm );
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].pxAttributeEventCallback =
vEnableNotification;
pxCharUUID.uu.uu16 = gattDemoCHAR_COUNTER_UUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xAttributeData.xUuid =
pxCharUUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xAttributeData.pucData =
NULL;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xAttributeData.xSize = 0;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xPermissions =
( eBTPermRead );
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xProperties =
( eBTPropRead | eBTPropNotify );
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].pxAttributeEventCallback =
vReadCounter;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xNbDescriptors = 1;
```
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].pxDescriptors[ 0 ] = &pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ];

xCharUUID.uu.uu16 = gattCHAR_CONTROL_UUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xAttributeData.xUuid = xCharUUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].pxDescriptors[ 0 ]
&pxGattDemoService->pxDescriptors[ egattDemoCharControlCCFGDESCR ];
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].xAttributeData.xUuid =
xCharUUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].xAttributeData.pucData = NULL;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].xAttributeData.xSize = 0;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].xPermissions =
{ eBTPermReadEncryptedMitm | eBTPermWriteEncryptedMitm };
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].xProperties =
{ eBTPropRead | eBTPropWrite };
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].pxAttributeEventCallback =
vWriteCommand;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].xNbDescriptors = 0;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounterCCFGDESCR ].pxDescriptors = NULL;

pxGattDemoService->xServiceType = eBTServiceTypePrimary;
pxGattDemoService->ucInstId = 0;

xStatus = BLE_AddService( pxGattDemoService );

3. Start the service.

xStatus = BLE_StartService( pxGattDemoService, vServiceStartedCb );

4. Subscribe to any event required for the service. In this example, we subscribe to a connection event.

xBleCallback.pxConnectionCb = vConnectionCallback;
BLE_RegisterEventCb( eBLEConnection, xCallback );

For full Amazon FreeRTOS BLE demo applications, see Bluetooth Low Energy Demo Applications.

Porting

User Input and Output Peripheral

A secure connection requires both input and output for numeric comparison. The
eBLENumericComparisonCallback event can be registered using the event manager:

xEvtCb.pxNumericComparisonCb = &prvNumericComparisonCb;
xBleStatus = BLE_RegisterEventCb( eBLENumericComparisonCallback, xEventCb );

The peripheral must display the numeric passkey and take the result of the comparison as an input.

Porting API Implementations

To port Amazon FreeRTOS to a new target, you must implement some APIs for the Wi-Fi Provisioning
service and BLE functionality.

Wi-Fi Provisioning APIs

To use the Wi-Fi provisioning service you must implement the following APIs:

• WIFI_NetworkGet
• WIFI_NetworkDelete
BLE APIs

To use the Amazon FreeRTOS BLE middleware, you must implement some APIs.

APIs Common Between GAP for Bluetooth Classic and GAP for BLE

- `pxBtManagerInit`
- `pxEnable`
- `pxDisable`
- `pxGetDeviceProperty`
- `pxSetDeviceProperty` (All options are mandatory except eBTpropertyRemoteRssi and eBTpropertyRemoteVersionInfo)
- `pxPair`
- `pxRemoveBond`
- `pxGetConnectionState`
- `pxPinReply`
- `pxSspReply`
- `pxGetTxpower`
- `pxGetLeAdapter`
- `pxDeviceStateChangedCb`
- `pxAdapterPropertiesCb`
- `pxSspRequestCb`
- `pxPairingStateChangedCb`
- `pxTxPowerCb`

APIs Specific to GAP for BLE

- `pxRegisterBleApp`
- `pxUnregisterBleApp`
- `pxBleAdapterInit`
- `pxStartAdv`
- `pxStopAdv`
- `pxSetAdvData`
- `pxConnParameterUpdateRequest`
- `pxRegisterBleAdapterCb`
- `pxAdvStartCb`
- `pxSetAdvDataCb`
- `pxConnParameterUpdateRequestCb`
- `pxCongestionCb`

GATT Server

- `pxRegisterServer`
- `pxUnregisterServer`
- `pxGattServerInit`
Mobile SDKs for Amazon FreeRTOS Bluetooth Devices

The Bluetooth Low Energy (BLE) Library is in public beta release for Amazon FreeRTOS and is subject to change.

You can use the Amazon FreeRTOS BLE Mobile SDKs to create mobile applications that interact with your microcontroller over BLE.

**Android SDK for Amazon FreeRTOS Bluetooth Devices**

Use the Amazon FreeRTOS BLE Android SDK to build Android mobile applications that interact with your microcontroller over BLE. For more information, see Amazon FreeRTOS BLE Mobile SDK for Android.

**Android SDK for Amazon FreeRTOS Bluetooth Devices**

Use the Amazon FreeRTOS BLE iOS SDK to build iOS mobile applications that interact with your microcontroller over BLE. For more information, see Amazon FreeRTOS BLE Mobile SDK for iOS.
Amazon FreeRTOS AWS IoT Device Defender Library

Overview

AWS IoT Device Defender is an AWS IoT service that allows you to audit the configuration of your devices, monitor connected devices to detect abnormal behavior, and to mitigate security risks. It gives you the ability to enforce consistent IoT configurations across your AWS IoT device fleet and respond quickly when devices are compromised.

Amazon FreeRTOS provides a library that allows your Amazon FreeRTOS-based devices to work with AWS IoT Device Defender. You can download the Amazon FreeRTOS Device Defender library using the Amazon FreeRTOS Console by adding the Device Defender library to your software configuration. You can also clone the Amazon FreeRTOS GitHub repository and find the library in the `lib` directory.

The source files for the Amazon FreeRTOS AWS IoT Device Defender library are located in `AmazonFreeRTOS/lib/defender`.

Source and Header Files

```
Amazon FreeRTOS
|-- lib
|   |-- defender
|   |   |-- # aws_defender.c
|   |   |-- # aws_defender_states.dot
|   |   |-- # aws_defender_states.png
|   |   |-- # draw_states.py
|   |   |-- # portable
|   |   |   |-- # freertos
|   |   |   |   |-- # aws_defender_cpu.c
|   |   |   |   |-- # aws_defender_tcp_conn.c
|   |   |   |   |-- # aws_defender_uptime.c
|   |   |   |-- # stub
|   |   |   |   |-- # aws_defender_cpu.c
|   |   |   |   |-- # aws_defender_tcp_conn.c
|   |   |   |   |-- # aws_defender_uptime.c
|   |   |   |-- # portable
|   |   |   |   |-- # freertos
|   |   |   |   |   |-- # aws_defender_cpu.c
|   |   |   |   |   |-- # aws_defender_tcp_conn.c
|   |   |   |   |   |-- # aws_defender_uptime.c
|   |   |   |   |-- # stub
|   |   |   |   |   |-- # aws_defender_cpu.c
|   |   |   |   |   |-- # aws_defender_tcp_conn.c
|   |   |   |   |   |-- # aws_defender_uptime.c
|   |   |   |   |   |-- # makefile
|   |   |   |-- # template
|   |   |   |   |-- # aws_defender_cpu.c
|   |   |   |   |-- # aws_defender_tcp_conn.c
|   |   |   |   |-- # aws_defender_uptime.c
|   |   |   |   |-- # makefile
|   |   |   |-- # unit_test
|   |   |   |   |-- # aws_defender_cpu.c
|   |   |   |   |-- # aws_defender_tcp_conn.c
|   |   |   |   |-- # aws_defender_uptime.c
|   |   |   |   |-- # makefile
|   |   |-- # report
|   |   |   |-- # aws_defender_report.c
|   |   |   |-- # aws_defender_report_cpu.c
|   |   |   |-- # aws_defender_report_header.c
|   |   |   |-- # aws_defender_report_tcp_conn.c
|   |   |   |-- # aws_defender_report_uptime.c
|   |   |-- include
|   |   |   |-- # aws_defender.h
|   |   |   |-- private
```
Developer Support

Amazon FreeRTOS Device Defender API Error Codes

eDefenderErrSuccess

The operation was successful.
eDefenderErrFailedToCreateTask

The operation could not be started.
eDefenderErrAlreadyStarted

The operation is already in progress.
eDefenderErrNotStarted

The Device Defender agent has not been started.
eDefenderErrOther

An unspecified error occurred.

Amazon FreeRTOS Device Defender API

This section contains information about the Device Defender API.

DEFENDER_MetricsInit

Specifies the Device Defender metrics your device will send to AWS IoT Device Defender.

```c
DefenderErr_t DEFENDER_MetricsInit(DefenderMetric_t * pxMetricsList);
```

Arguments

metrics_list

A list of Device Defender metrics. Valid values are:

- DEFENDER_tcp_connections - tracks the number of TCP connections.

Return Value

Returns one of the DefenderErr_t enums. For more information, see Amazon FreeRTOS Device Defender API Error Codes (p. 83).
**DEFENDER_ReportPeriodSet**

Sets the report period interval in seconds. Device Defender provides metric reports on an interval. If the device is awake, and the interval has elapsed, the device reports the metrics.

```c
DefenderErr_t DEFENDER_ReportPeriodSet(int32_t LPeriodSec);
```

**Arguments**

- `period_sec`
  - The number of seconds after which a report is sent to AWS IoT Device Defender.

**Return Value**

Returns one of the `DefenderErr_t` enums. For more information, see Amazon FreeRTOS Device Defender API Error Codes (p. 83).

**DEFENDER_Start**

Starts the Device Defender agent.

```c
DefenderErr_t DEFENDER_Start(void);
```

**Return Value**

Returns one of the `DefenderErr_t` enums. For more information, see Amazon FreeRTOS Device Defender API Error Codes (p. 83).

**DEFENDER_Stop**

Stops the Device Defender agent.

```c
DefenderErr_t DEFENDER_Stop(void);
```

**Return Value**

Returns one of the `DefenderErr_t` enums. For more information, see Amazon FreeRTOS Device Defender API Error Codes (p. 83).

**DEFENDER_ReportStatusGet**

Gets the status of the last Device Defender report. Valid status code values are:

- `eDefenderRepSuccess`
  - The last report was successfully sent and acknowledged.
- `eDefenderRepInit`
  - Device Defender has been started, but no report has been sent.
Example Usage

Using Device Defender in Your Embedded Application

The following code shows how to configure and start the Device Defender agent from your embedded application:

```c
void MyDefenderInit(void)
{
    // Specify metrics to send to Device Defender
    defender_metric_t metrics_list[] = {
        DEFENDER_tcp_connections
    };
    ( void ) DEFENDER_MetricsInit( metrics_list );

    // Set the reporting interval
    // You can use a shorter period to trigger the violation faster, however
    // the Device Defender service is not guaranteed to accept reports faster
    // than every 300 seconds (5 minutes) per device.
    int report_period_sec = 300;
    ( void ) DEFENDER_ReportPeriodSet( report_period_sec );

    // Start the Device Defender agent
    DEFENDER_Start();
}
```

Amazon FreeRTOS AWS IoT Greengrass Discovery Library

Overview

The AWS IoT Greengrass Discovery library is used by your microcontroller devices to discover a Greengrass core on your network. Using the AWS IoT Greengrass Discovery APIs, your device can send messages to a Greengrass core after it finds the core's endpoint.

The source files for the Amazon FreeRTOS AWS IoT Greengrass library are located in `AmazonFreeRTOS/lib/greengrass`.

Dependencies and Requirements

To use the Greengrass Discovery library, you must create a thing in AWS IoT, including a certificate and policy. For more information, see AWS IoT Getting Started. You must set values for the following constants in the `AmazonFreeRTOS/demos/common/include/aws_client_credentials.h` file:
clientcredentialMQTT_BROKER_ENDPOINT

Your AWS IoT endpoint.

clientcredentialIOT_THING_NAME

The name of your IoT thing.

clientcredentialWIFI_SSID

The SSID for your Wi-Fi network.

clientcredentialWIFI_PASSWORD

Your Wi-Fi password.

clientcredentialWIFI_SECURITY

The type of security used by your Wi-Fi network.

keyCLIENT_CERTIFICATE_PEM

The certificate PEM associated with your thing.

keyCLIENT_PRIVATE_KEY_PEM

The private key PEM associated with your thing.

You must have a Greengrass group and core device set up in the console. For more information, see Getting Started with AWS IoT Greengrass.

Although the MQTT library is not required for Greengrass connectivity, we strongly recommend you install it. The library can be used to communicate with the Greengrass core after it has been discovered.

Source and Header Files

| Amazon FreeRTOS |
| --- | --- |
| + lib |
| + greengrass |
| + # aws_greengrass_discovery.c |
| + # aws_helper_secure_connect.c |
| + include |
| + # aws_greengrass_discovery.h |
| + private |
| + aws_ggd_config_defaults.h |

API Reference

For a full API reference, see Greengrass API Reference.

Example Usage

Greengrass Workflow

The MCU device initiates the discovery process by requesting from AWS IoT a JSON file that contains the Greengrass core connectivity parameters. There are two methods for retrieving the Greengrass core connectivity parameters from the JSON file:

- Automatic selection iterates through all of the Greengrass cores listed in the JSON file and connects to the first one available.
• Manual selection uses the information in `aws_ggd_config.h` to connect to the specified Greengrass core.

**How to Use the Greengrass API**

All default configuration options for the Greengrass API are defined in `lib\include\private\aws_ggd_config_defaults.h`. You can override any of these settings in `lib\include\private`.

If only one Greengrass core is present, call `GGD_GetGGCIPandCertificate` to request the JSON file with Greengrass core connectivity information. When `GGD_GetGGCIPandCertificate` is returned, the `pcBuffer` parameter contains the text of the JSON file. The `pxHostAddressData` parameter contains the IP address and port of the Greengrass core to which you can connect.

For more customization options, like dynamically allocating certificates, you must call the following APIs:

- **GGD_JSONRequestStart**
  
  Makes an HTTP GET request to AWS IoT to initiate the discovery request to discover a Greengrass core. `GGD_SecureConnect_Send` is used to send the request to AWS IoT.

- **GGD_JSONRequestGetSize**
  
  Gets the size of the JSON file from the HTTP response.

- **GGD_JSONRequestGetFile**
  
  Gets the JSON object string. `GGD_JSONRequestGetSize` and `GGD_JSONRequestGetFile` use `GGD_SecureConnect_Read` to get the JSON data from the socket. `GGD_JSONRequestStart`, `GGD_SecureConnect_Send`, `GGD_JSONRequestGetSize` must be called to receive the JSON data from AWS IoT.

- **GGD_GetIPandCertificateFromJSON**
  
  Extracts the IP address and the Greengrass core certificate from the JSON data. You can turn on automatic selection by setting the `xAutoSelectFlag` to `True`. Automatic selection finds the first core device your FreeRTOS device can connect to. To connect to a Greengrass core, call the `GGD_SecureConnect_Connect` function, passing in the IP address, port, and certificate of the core device. To use manual selection, set the following fields of the `HostParameters_t` parameter:

  - **pcGroupName**
    
    The ID of the Greengrass group to which the core belongs. You can use the `aws greengrass list-groups` CLI command to find the ID of your Greengrass groups.

  - **pcCoreAddress**
    
    The ARN of the Greengrass core to which you are connecting.

**Amazon FreeRTOS MQTT Library (Beta)**

The new MQTT library is in public beta release for Amazon FreeRTOS and is subject to change.

**Overview**

You can use the Amazon FreeRTOS MQTT library to create applications that publish and subscribe to MQTT topics, as MQTT clients on a network. The Amazon FreeRTOS MQTT library implements the MQTT
3.1.1 standard for compatibility with the AWS IoT MQTT server. The library is also compatible with other MQTT servers.

The source files for the Amazon FreeRTOS MQTT library are located in AmazonFreeRTOS/lib/mqtt.

The Amazon FreeRTOS MQTT library documented here is in public beta. For more information about the legacy Amazon FreeRTOS MQTT library, see Amazon FreeRTOS MQTT Library (Legacy) (p. 90).

Dependencies and Requirements

The Amazon FreeRTOS MQTT library has the following dependencies:

- The queue library for maintaining the data structures that manage in-progress MQTT operations
- The logging library, if the configuration parameter AWS_IOT_MQTT_LOG_LEVEL is not set to AWS_IOT_LOG_NONE
- The platform layer that provides an interface to the operating system for thread management, clock functions, networking, and other platform-level functionality
- C standard library headers

The diagram below illustrates these dependencies.

![Diagram](image)

Features

The Amazon FreeRTOS MQTT library has the following features:

- By default, the library has a fully asynchronous MQTT API. You can opt to use the library synchronously with the AwsIotMqtt_Wait function.
- The library is thread-aware and parallelizable for high throughput.
- The library features scalable performance and footprint. Use the configuration setting to tailored the library to a system's resources.

Configuration

Configuration settings for the Amazon FreeRTOS MQTT library are defined as C proprocessor constants. Set configuration settings as #define constants in a file named AWS_IOT_CONFIG_FILE, or by using a compiler option such as `-D` in gcc. Because configuration settings are defined as compile-time constants, a library must be rebuilt if a configuration setting is changed. The MQTT library uses default values when configuration settings are not defined.
For more information about configuring the Amazon FreeRTOS MQTT library, see MQTT API Reference (Beta).

**API Reference**

For a full API reference, see MQTT API Reference (Beta).

**Example Usage**

`aws_iot_demo_mqtt.c`

For example usage of the Amazon FreeRTOS MQTT library, see MQTT demo application defined in `aws_iot_demo_mqtt.c`.

The MQTT demo demonstrates the subscribe-publish workflow of MQTT. After subscribing to multiple topic filters, the application publishes bursts of data to various topic names. As each message arrives, the demo publishes an acknowledgement message back to the MQTT server.

To run the MQTT demo, you need to configure the following parameters:

**Global Demo Configuration Parameters**

These configuration parameters apply to all demos.

`AWS_IOT_DEMO_SECURED_CONNECTION`

Determines if the demo uses a TLS-secured connection with the remote host by default.

`AWS_IOT_DEMO_SERVER`

The default remote host to use.

`AWS_IOT_DEMO_PORT`

The default remote port to use.

`AWS_IOT_DEMO_ROOT_CA`

The path to the default trusted server root certificate to use.

`AWS_IOT_DEMO_CLIENT_CERT`

The path to the default client certificate to use.

`AWS_IOT_DEMO_PRIVATE_KEY`

The path to the default client certificate private key to use.

**MQTT Demo Configuration Parameters**

These configuration parameters apply to the MQTT demo.

`AWS_IOT_DEMO_MQTT_PUBLISH_BURST_SIZE`

The number of messages to publish in each burst.

`AWS_IOT_DEMO_MQTT_PUBLISH_BURST_COUNT`

The number of publish bursts in this demo.
Amazon FreeRTOS MQTT Library (Legacy)

Overview

Amazon FreeRTOS includes an open source MQTT client library that you can use to create applications that publish and subscribe to MQTT topics, as MQTT clients on a network.

The source files for the Amazon FreeRTOS MQTT library are located in AmazonFreeRTOS/lib/mqtt.

A new Amazon FreeRTOS MQTT Library is in public beta. For more information, see Amazon FreeRTOS MQTT Library (Beta) (p. 87).

The FreeRTOS MQTT Agent

Amazon FreeRTOS also includes an open source daemon, called the FreeRTOS MQTT agent, that manages the MQTT library for you. The MQTT agent provides a simple interface to connect, publish, and subscribe to MQTT topics with the underlying MQTT library.

The MQTT agent runs in a separate FreeRTOS task and automatically sends regular keep-alive messages, as documented by the MQTT protocol specification. All the MQTT APIs are blocking and take a timeout parameter, which is the maximum amount of time the API waits for the corresponding operation to complete. If the operation does not complete in the provided time, the API returns timeout error code.

Dependencies and Requirements

The Amazon FreeRTOS MQTT library uses the Amazon FreeRTOS Secure Sockets Library (p. 97) and the Amazon FreeRTOS Buffer Pool library. If the MQTT agent connects to a secure MQTT broker, the library also uses the Amazon FreeRTOS Transport Layer Security (TLS) (p. 103).

Features

Callback

You can specify an optional callback that is invoked whenever the MQTT agent is disconnected from the broker or whenever a publish message is received from the broker. The received publish message is stored in a buffer taken from the central buffer pool. This message is passed to the callback. This callback runs in the context of the MQTT task and therefore must be quick. If you need to do longer processing, you must take the ownership of the buffer by returning pdTRUE from the callback. You must then return the buffer back to the pool whenever you are done by calling FreeRTOS_Agent_ReturnBuffer.

Subscription Management

Subscription management enables you to register a callback per subscription filter. You supply this callback while subscribing. It is invoked whenever a publish message received on a topic matches the subscribed topic filter. The buffer ownership works the same way as described in the case of generic callback.

MQTT Task Wakeup

MQTT task wakeup wakes up whenever the user calls an API to perform any operation or whenever a publish message is received from the broker. This asynchronous wakeup upon receipt of a publish message is possible on platforms that are capable of informing the host MCU about the data received on a connected socket. Platforms that do not have this capability require the MQTT task to continuously
poll for the received data on the connected socket. To ensure the delay between receiving a publish message and invoking the callback is minimal, the `mqttconfigMQTT_TASK_MAX_BLOCK_TICKS` macro controls the maximum time an MQTT task can remain blocked. This value must be short for the platforms that lack the capability to inform the host MCU about received data on a connected socket.

### Source and Header Files

```
Amazon FreeRTOS
|-- lib
| |-- mqtt
|   |-- aws_mqtt_lib.c [Required to use the MQTT library and the MQTT agent]
|   |-- aws_mqtt_agent.c [Required to use the MQTT agent]
| |-- include
|   |-- aws_doubly_linked_list.h
|   |-- aws_mqtt_agent_config_defaults.h
|   |-- aws_mqtt_buffer.h
|   |-- aws_mqtt_config_defaults.h
|   |-- aws_mqtt_agent.h [Include to use the MQTT agent API]
|   |-- aws_mqtt_lib.h [Include to use the MQTT library API]
```

### Major Configurations

These flags can be specified during the MQTT connection request:

- `mqttconfigKEEP_ALIVE_ACTUAL_INTERVAL_TICKS`: The frequency of the keep-alive messages sent.
- `mqttconfigENABLE_SUBSCRIPTION_MANAGEMENT`: Enable subscription management.
- `mqttconfigMAX_BROKERS`: Maximum number of simultaneous MQTT clients.
- `mqttconfigMQTT_TASK_STACK_DEPTH`: The task stack depth.
- `mqttconfigMQTT_TASK_PRIORITY`: The priority of the MQTT task.
- `mqttconfigRX_BUFFER_SIZE`: Length of the buffer used to receive data.
- `mqttagentURL_IS_IP_ADDRESS`: Set this bit in xFlags if the provided URL is an IP address.
- `mqttagentREQUIRE_TLS`: Set this bit in xFlags to use TLS.
- `mqttagentUSE_AWS_IOT_ALPN_443`: Set this bit in xFlags to use AWS IoT support for MQTT over TLS port 443.

For more information about ALPN, see the AWS IoT Protocols in the AWS IoT Developer Guide and the MQTT with TLS Client Authentication on Port 443: Why It Is Useful and How It Works blog post on the Internet of Things on AWS blog.

### Optimization

**Processing Received Packets Without Delay**

The task that implements the MQTT agent spends most of its time in the Blocked state (so not using any CPU cycles) waiting for events to process. MQTT throughput is maximized by unblocking the agent task as soon as an MQTT packet is received from the network. If that is done the received packet is processed
at the earliest opportunity. If that is not done the received packet will not be processed until the MQTT agent leaves the Blocked state for another reason.

The MQTT agent is removed from the Blocked state by the execution of a callback that is installed by the MQTT agent calling SOCKETS_SetSockOpt() with the lOptionName parameter set to SOCKETS_SO_WAKEUP_CALLBACK. Links to the secure sockets documentation are needed here. If you are using the FreeRTOS+TCP TCP/IP stack the callback is executed at the correct time provided ipconfigSOCKET_HAS_USER_WAKE_CALLBACK is set to 1 in FreeRTOSIPConfig.h (which is the TCP/IP stack’s configuration file). If you are not using the FreeRTOS+TCP TCP/IP stack then the secure sockets ensure this functionality is included in your implementation of the secure sockets abstraction layer for the stack in use.

If the TCP/IP stack cannot unblock the MQTT agent as soon as data is received then the maximum time between a packet being received and the packet being processed is set by the mqttconfigMQTT_TASK_MAX_BLOCK_TICKS constant.

**Minimizing RAM Consumption**

The following configuration constants directly affect the amount of RAM required by the MQTT agent:

- mqttconfigMQTT_TASK_STACK_DEPTH
- mqttconfigMAX_BROKERS
- mqttconfigMAX_PARALLEL_OPS
- mqttconfigRX_BUFFER_SIZE

You should set these constants to the minimum values possible.

**Requirements and Usage Restrictions**

The MQTT agent task is created using the xTaskCreateStatic() API function - so the task's stack and control block are statically allocated at compile time. That ensures the MQTT agent can be used in applications that do not allow dynamic memory allocation, but does mean there is a dependency on configSUPPORT_STATIC_ALLOCATION being set to 1 in FreeRTOSConfig.h.

The MQTT agent uses the FreeRTOS direct to task notification feature. Calling an MQTT agent API function may change the calling task's notification value and state.

MQTT packets are stored in buffers provided by the Buffer Pool module. It is highly recommended to ensure the number of buffers in the pool is at least double the number of MQTT transactions that will be in progress at any one time.

**Developer Support**

mqttconfigASSERT

mqttconfigASSERT() is equivalent to, and used in exactly the same way as, the FreeRTOS configASSERT() macro. If you want assert statements in the MQTT agent then define mqttconfigASSERT(). If you do not want assert statements in the MQTT agent then leave mqttconfigASSERT() undefined. If you define mqttconfigASSERT() to call the FreeRTOS configASSERT(), as shown below, then the MQTT agent will only include assert statements if the FreeRTOS configASSERT() is defined.

#define mqttconfigASSERT( x ) configASSERT( x )

mqttconfigENABLE_DEBUG_LOGS
Set `mqttconfigENABLE_DEBUG_LOGS` to `1` to print debug logs via calls to `vLoggingPrintf()`.

**Initialization**

Both the MQTT agent and its dependent libraries must be initialized, as shown below, before attempting MQTT communication. Initialize the libraries after a network connection is established.

```c
BaseType_t SYSTEM_Init() { BaseType_t xResult = pdPASS; /* The bufferpool libraries provide the buffers use to store MQTT packets.*/
    xResult = BUFFERPOOL_Init();
    if( xResult == pdPASS ) { /* Create the MQTT agent task. */
        xResult = MQTT_AGENT_Init();
        if( xResult == pdPASS ) { /* Initialize the secure sockets abstraction layer.*/
            xResult = SOCKETS_Init();
        }
    }
    return xResult; }
```

**API Reference**

For a full API reference, see MQTT Library API Reference (Legacy) and MQTT Agent API Reference (Legacy).

**Porting**

The Secure Sockets abstraction layer that the MQTT agent calls must be ported to specific architectures. For more information, see the Amazon FreeRTOS Porting Guide.

**Amazon FreeRTOS Over-the-Air (OTA) Agent Library**

**Overview**

The OTA agent enables you to manage the notification, download, and verification of firmware updates for Amazon FreeRTOS devices. By using the OTA agent library, you can logically separate firmware updates and the application running on your devices. The OTA agent can share a network connection with the application. By sharing a network connection, you can potentially save a significant amount of RAM. In addition, the OTA agent library allows you to define application-specific logic for testing, committing, or rolling back a firmware update.

The source files for the Amazon FreeRTOS OTA agent library are located in `AmazonFreeRTOS/lib/ota`.

For more information about using Over-the-Air updates with Amazon FreeRTOS, see Amazon FreeRTOS Over-the-Air Updates (p. 108).

**Features**

Here is the complete OTA agent interface:

- **OTA_AgentInit**
  
  Initializes the OTA agent. The caller provides messaging protocol context, an optional callback, and a timeout.

- **OTA_AgentShutdown**
  
  Cleans up resources after using the OTA agent.

- **OTA_GetAgentState**
  
  Gets the current state of the OTA agent.
OTA_ActivateNewImage

Activates the newest microcontroller firmware image received through OTA. (The detailed job status should now be self-test.)

OTA_SetImageState

Sets the validation state of the currently running microcontroller firmware image (testing, accepted or rejected).

OTA_GetImageState

Gets the state of the currently running microcontroller firmware image (testing, accepted or rejected).

OTA_CheckForUpdate

Requests the next available OTA update from the OTA Update service.

Source and Header Files

```
Amazon FreeRTOS
|-- lib
|  |-- ota
|  |  |-- aws_ota_agent.c
|  |  |-- aws_ota_cbor.c
|  |  |-- portable
|  |  |  |-- README.md
|  |  |-- vendor
|  |  |  |-- board
|  |  |  |  |-- aws_ota_pal.c
|  |  |-- include
|  |  |  |-- aws_ota_agent.h
|  |  |-- private
|  |  |  |-- aws_ota_agent_internal.h
|  |  |  |-- aws_ota_cbor.h
|  |  |  |-- aws_ota_cbor_internal.h
|  |  |  |-- aws_ota_pal.h
|  |  |  |-- aws_ota_types.h
```

API Reference

For a full API reference, see OTA Agent API Reference.

Example Usage

A typical OTA-capable device application drives the OTA agent using the following sequence of API calls:

1. Connect to the AWS IoT MQTT broker. For more information, see Amazon FreeRTOS MQTT Library (Legacy) (p. 90).

2. Initialize the OTA agent by calling OTA_AgentInit. Your application may define a custom OTA callback function or use the default callback by specifying a NULL callback function pointer. You must also supply an initialization timeout.

   The callback implements application-specific logic that executes after completing an OTA update job. The timeout defines how long to wait for the initialization to complete.

3. If OTA_AgentInit timed out before the agent was ready, you can call OTA_GetAgentState to confirm that the agent is initialized and operating as expected.
4. When the OTA update is complete, Amazon FreeRTOS calls the job completion callback with one of the following events: accepted, rejected, or self test.

5. If the new firmware image has been rejected (for example, due to a validation error), the application can typically ignore the notification and wait for the next update.

6. If the update is valid and has been marked as accepted, call `OTA_ActivateNewImage` to reset the device and boot the new firmware image.

Porting

For information about porting OTA functionality to your platform, see OTA Portable Abstraction Layer.

Amazon FreeRTOS Public Key Cryptography Standard (PKCS) #11 Library

Overview

Public Key Cryptography Standard #11 (PKCS#11) is a cryptographic API that abstracts key storage, get/set properties for cryptographic objects, and session semantics. See `pkcs11.h` (obtained from OASIS, the standard body) in the Amazon FreeRTOS source code repository. In the Amazon FreeRTOS reference implementation, PKCS#11 API calls are made by the TLS helper interface in order to perform TLS client authentication during `SOCKETS_Connect`. PKCS#11 API calls are also made by our one-time developer provisioning workflow to import a TLS client certificate and private key for authentication to the AWS IoT MQTT broker. Those two use cases, provisioning and TLS client authentication, require implementation of only a small subset of the PKCS#11 interface standard.

The source files for the Amazon FreeRTOS PKCS#11 library are located in `AmazonFreeRTOS/lib/secure_sockets/portable`.

Features

The following subset of PKCS#11 is used. This list is in roughly the order in which the routines are called in support of provisioning, TLS client authentication, and cleanup. For detailed descriptions of the functions, see the PKCS#11 documentation provided by the standard committee.

Provisioning API

- `C_GetFunctionList`
- `C_Initialize`
- `C_CreateObject CKO_PRIVATE_KEY` (for device private key)
- `C_CreateObject CKO_CERTIFICATE` (for device certificate and code verification certificate)
- `C.GenerateKeyPair`

Client Authentication

- `C_Initialize`
- `C_GetSlotList`
- `C_OpenSession`
- `C_FindObjectsInit`
- `C_FindObjects`
Asymmetric Cryptosystem Support

The Amazon FreeRTOS PKCS#11 reference implementation supports 2048-bit RSA (signing only) and ECDSA with the NIST P-256 curve. The following instructions describe how to create an AWS IoT thing based on a P-256 client certificate.

Make sure you are using the following (or more recent) versions of the AWS CLI and OpenSSL:

```
aws --version
aws-cli/1.11.176 Python/2.7.9 Windows/8 botocore/1.7.34
openssl version
OpenSSL 1.0.2g  1 Mar 2016
```

The following steps are written with the assumption that you have used the `aws configure` command to configure the AWS CLI.

Creating an AWS IoT thing based on a P-256 client certificate

1. Run `aws iot create-thing --thing-name dcgecc` to create an AWS IoT thing.
2. Run `openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt ec_param_enc:named_curve -outform PEM -out dcgecc.key` to use OpenSSL to create a P-256 key.
3. Run `openssl req -new -nodes -days 365 -key dcgecc.key -out dcgecc.req` to create a certificate enrollment request signed by the key created in step 2.
4. Run `aws iot create-certificate-from-csr --certificate-signing-request file://dcgecc.req --set-as-active --certificate-pem-outfile dcgecc.crt` to submit the certificate enrollment request to AWS IoT.
5. Run `aws iot attach-thing-principal --thing-name dcgecc --principal "arn:aws:iot:us-east-1:123456789012:cert/86e41339a6d1b2c67abf31f455092c0debff021ffbc67c4d238d1326c7de4"` to attach the certificate (referenced by the ARN output by the previous command) to the thing.
6. Run `aws iot create-policy --policy-name FullControl --policy-document file://policy.json` to create a policy. (This policy is too permissive. It should be used for development purposes only.)

The following is a listing of the policy.json file specified in the `create-policy` command. You can omit the `greengrass:*` action if you don't want to run the Amazon FreeRTOS demo for Greengrass connectivity and discovery.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:*",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "greengrass:*",
            "Resource": "*"
        }
    ]
}
```

7. Run `aws iot attach-principal-policy --policy-name FullControl --principal "arn:aws:iot:us-east-1:785484208847:cert/86e41339a6d1bbc67af31fafa455092cdebf8f21ffbc67c4d238d1326c7de..."` to attach the principal (certificate) and policy to the thing.

Now, follow the steps in the AWS IoT Getting Started section of this guide. Don't forget to copy the certificate and private key you created into your `aws_clientcredential_keys.h` file. Copy your thing name into `aws_clientcredential.h`.

Amazon FreeRTOS Secure Sockets Library

Overview

You can use the Amazon FreeRTOS Secure Sockets library to create embedded applications that communicate securely. The library is designed to make onboarding easy for software developers from various network programming backgrounds.

The Amazon FreeRTOS Secure Sockets library is based on the Berkeley sockets interface, with an additional secure communication option by TLS protocol. For information about the differences between the Amazon FreeRTOS Secure Sockets library and the Berkeley sockets interface, see `SOCKETS_SetSockOpt` in the Secure Sockets API Reference.

The source files for the Amazon FreeRTOS Secure Sockets library are located in `AmazonFreeRTOS/lib/secure_sockets/portable`.

Note

Currently, only client APIs are supported for Amazon FreeRTOS Secure Sockets.

Dependencies and Requirements

The Amazon FreeRTOS Secure Sockets library depends on a TCP/IP stack and on a TLS implementation. Ports for Amazon FreeRTOS meet these dependencies in one of three ways:

- A custom implementation of both TCP/IP and TLS
- A custom implementation of TCP/IP, and the Amazon FreeRTOS TLS layer with mbedTLS
• **FreeRTOS+TCP** and the Amazon FreeRTOS TLS layer with **mbedTLS**

The dependency diagram below shows the reference implementation included with the Amazon FreeRTOS Secure Sockets library. This reference implementation supports TLS and TCP/IP over Ethernet and Wi-Fi with FreeRTOS+TCP and mbedTLS as dependencies. For more information about the Amazon FreeRTOS TLS layer, see Amazon FreeRTOS Transport Layer Security (TLS) (p. 103).

**Features**

Amazon FreeRTOS Secure Sockets library features include:

- A standard, Berkeley Sockets-based interface
- Thread-safe APIs for sending and receiving data
- Easy-to-enable TLS

**Footprint**

**Code Size (example generated with GCC for ARM Cortex-M)**

<table>
<thead>
<tr>
<th>File name</th>
<th>Size (optimized for speed)</th>
<th>Size (optimized for speed and size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure Sockets Library</td>
<td>Varies by port</td>
<td>Varies by port</td>
</tr>
<tr>
<td>For example, for the TI CC3220SF:</td>
<td>5.0 K</td>
<td>4.3 K</td>
</tr>
<tr>
<td>lib/secure_sockets/portable/ti/cc3220_launchpad/aws_secure_sockets.c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Source and Header Files

```
Amazon FreeRTOS
+ # lib
  + # include
    + # aws_secure_sockets.h
    + # private
      + # aws_secure_sockets_config_defaults.h
+ # secure_sockets
  + - portable
    + - ...
  + # aws_secure_sockets.c
```

Troubleshooting

Error codes

The error codes that the Amazon FreeRTOS Secure Sockets library returns are negative values. For more information about each error code, see Secure Sockets Error Codes in the Secure Sockets API Reference.

**Note**

If the Amazon FreeRTOS Secure Sockets API returns an error code, the Amazon FreeRTOS MQTT Library (Legacy) (p. 90), which depends on the Amazon FreeRTOS Secure Sockets library, returns the error code AWS_IOT_MQTT_SEND_ERROR.

Developer Support

The Amazon FreeRTOS Secure Sockets library includes two helper macros for handling IP addresses:

**SOCKETS_inet_addr_quick**

This macro converts an IP address that is expressed as four separate numeric octets into an IP address that is expressed as a 32-bit number in network-byte order.

**SOCKETS_inet_ntoa**

This macro converts an IP address that is expressed as a 32-bit number in network byte order to a string in decimal-dot notation.

Usage Restrictions

Only TCP sockets are supported by the Amazon FreeRTOS Secure Sockets library. UDP sockets are not supported.

Only client APIs are supported by the Amazon FreeRTOS Secure Sockets library. Server APIs, including `Bind`, `Accept`, and `Listen`, are not supported.

Initialization

To use the Amazon FreeRTOS Secure Sockets library, you need to initialize the library and its dependencies. To initialize the Secure Sockets library, use the following code in your application:

```
BaseType_t xResult = pdPASS;
xResult = SOCKETS_Init();
```

Dependent libraries must be initialized separately. For example, if FreeRTOS+TCP is a dependency, you need to invoke `FreeRTOS_IPInit` in your application as well.
API Reference

For a full API reference, see Secure Sockets API Reference.

Example Usage

The following code connects a client to a server.

```c
#include "aws_secure_sockets.h"
#define configSERVER_ADDR0                     127
#define configSERVER_ADDR1                     0
#define configSERVER_ADDR2                     0
#define configSERVER_ADDR3                     1
#define configCLIENT_PORT                      443

/* Rx and Tx timeouts are used to ensure the sockets do not wait too long for
* missing data. */
static const TickType_t xReceiveTimeOut = pdMS_TO_TICKS( 2000 );
static const TickType_t xSendTimeOut = pdMS_TO_TICKS( 2000 );

/* PEM-encoded server certificate */
/* The certificate used below is one of the Amazon Root CAs.
Change this to the certificate of your choice. */
static const char cTlsECHO_SERVER_CERTIFICATE_PEM[] =
"-----BEGIN CERTIFICATE-----
"MIIBtjCCAVugAwIBAgITBmyf1XSXMy/Onwa2iedgPySjAKBgggghkjOPQQDAjA5
"MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBbWdWb25j
"Um9vdCBDQSAzMB4XDTE1MDUyNjAwMDAwMFoXDTQwMDUyNjAwMDAwMFowOTELMAkG
"A1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEMBIGA1UEAxMQQW1hem9uIFJvb3Qg
"Q0EgMzBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCmXp8ZBf8ANm+gBG1dG8lKl
"ui2yEuJSLtf6ycXYqm0fc4E705hrOxw2pcVOho6AF2hiRVd9RFgdsfzlZwjrT66
"QjBAMA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/BAQDAgGGMBoGA1udDgQWBB
"ttvXBP43rDCGB5FxwzS1GbF4wDAKgKhkjOPQQDAgNJADBGAI4IWSoxe3jfskr
"BqWTrBqTaGFy/wGHRQsceGCMqSuMuMQCIQCCAu/xJyz1vnxr4tiz+OpAUFtM
"YYB1INH8wfdYoOw==
"-----END CERTIFICATE-----
";
static const uint32_t ulTlsECHO_SERVER_CERTIFICATE_LENGTH =
sizeof( cTlsECHO_SERVER_CERTIFICATE_PEM );

void vConnectToServerWithSecureSocket( void )
{
    Socket_t xSocket;
    SocketsSockaddr_t xEchoServerAddress;
    BaseType_t xTransmitted, lStringLength;

    /* Create a TCP socket. */
xSocket = SOCKETS_Socket( SOCKETS_AF_INET, SOCKETS_SOCK_STREAM, SOCKETS_IPPROTO_TCP );
configASSERT( xSocket != SOCKETS_INVALID_SOCKET );

    /* Set a timeout so a missing reply does not cause the task to block indefinitely. */
    SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_RCVTIMEO, &xReceiveTimeOut,
sizeof( xReceiveTimeOut ) );
    SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_SNDTIMEO, &xSendTimeOut,
sizeof( xSendTimeOut ) );
```

100
/* Set the socket to use TLS */
SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_REQUIRE_TLS, NULL, (size_t) 0 );
SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_TRUSTED_SERVER_CERTIFICATE,
cTlsECHO_SERVER_CERTIFICATE_PEM, ulTlsECHO_SERVER_CERTIFICATE_LENGTH );

if( SOCKETS_Connect( xSocket, &xEchoServerAddress, sizeof( xEchoServerAddress ) ) == 0 )
{
  /* Send the string to the socket. */
  xTransmitted = SOCKETS_Send( xSocket, /* The socket receiving. */
                               (void *) "some message", /* The data being sent. */
                               12, /* The length of the data being sent. */
                               0 ); /* No flags. */

  if( xTransmitted < 0 )
  {
    /* Error while sending data*/
    return;
  }

  SOCKETS_Shutdown( xSocket, SOCKETS_SHUT_RDWR );
} else
{
  //failed to connect to server
}

SOCKETS_Close( xSocket );

For a full example, see the Secure Sockets Echo Client Demo.

Porting

Amazon FreeRTOS Secure Sockets depends on a TCP/IP stack and on a TLS implementation. Depending on your stack, to port the Secure Sockets library, you might need to port some of the following:

- The FreeRTOS+TCP TCP/IP stack
- The Amazon FreeRTOS Public Key Cryptography Standard (PKCS) #11 Library (p. 95)
- The Amazon FreeRTOS Transport Layer Security (TLS) (p. 103)

For more information about porting, see the Amazon FreeRTOS Qualification Program Developer Guide and the Amazon FreeRTOS Porting Guide.

Amazon FreeRTOS AWS IoT Device Shadow Library

Overview

The Amazon FreeRTOS device shadow APIs define functions to create, update, and delete device shadows. For more information about Amazon FreeRTOS device shadows, see Device Shadows. Device shadows are accessed using the MQTT protocol. The FreeRTOS device shadow API works with the MQTT API and handles the details of working with the MQTT protocol.

The source files for the Amazon FreeRTOS AWS IoT device shadow library are located in AmazonFreeRTOS/lib/shadow.
Dependencies and Requirements

To use AWS IoT Device Shadows with Amazon FreeRTOS, you need to create a thing in AWS IoT, including a certificate and policy. For more information, see AWS IoT Getting Started. You must set values for the following constants in the AmazonFreeRTOS/demos/common/include/aws_client_credentials.h file:

- **clientcredentialMQTT_BROKER_ENDPOINT**
  Your AWS IoT endpoint.

- **clientcredentialIOT_THING_NAME**
  The name of your IoT thing.

- **clientcredentialWIFI_SSID**
  The SSID of your Wi-Fi network.

- **clientcredentialWIFI_PASSWORD**
  Your Wi-Fi password.

- **clientcredentialWIFI_SECURITY**
  The type of Wi-Fi security used by your network.

- **keyCLIENT_CERTIFICATE_PEM**
  The certificate PEM associated with your IoT thing. For more information, see Configure Your Project (p. 10).

- **keyCLIENT_PRIVATE_KEY_PEM**
  The private key PEM associated with your IoT thing. For more information, see Configure Your Project (p. 10).

Make sure the Amazon FreeRTOS MQTT library is installed on your device. For more information, see Amazon FreeRTOS MQTT Library (Legacy) (p. 90). Make sure that the MQTT buffers are large enough to contain the shadow JSON files. The maximum size for a device shadow document is 8 KB. All default settings for the device shadow API can be set in the lib\include\private\aws_shadow_config_defaults.h file. You can modify any of these settings in the demos/<platform>/common/config_files/aws_shadow_config.h file.

You must have an IoT thing registered with AWS IoT, including a certificate with a policy that permits accessing the device shadow. For more information, see AWS IoT Getting Started.

Source and Header Files

```
Amazon FreeRTOS
|+
|  - lib
|    |
|    + - shadow
|        |
|        + # aws_shadow.c
|        |
|        + # aws_shadow_json.c
|        + - include
|            + - aws_shadow.h
|            + - private
|                + - aws_shadow_config_defaults.h
|                + - aws_shadow_json.h
```
### API Reference

For a full API reference, see [Device Shadow API Reference](#).

### Example Usage

1. Use the `SHADOW_ClientCreate` API to create a shadow client. For most applications, the only field to fill is `xCreateParams.xMQTTClientType = eDedicatedMQTTClient`.
2. Establish an MQTT connection by calling the `SHADOW_ClientConnect` API, passing the client handle returned by `SHADOW_ClientCreate`.
3. Call the `SHADOW_RegisterCallbacks` API to configure callbacks for shadow update, get, and delete.

After a connection is established, you can use the following APIs to work with the device shadow:

- `SHADOW_Delete`
  - Delete the device shadow.
- `SHADOW_Get`
  - Get the current device shadow.
- `SHADOW_Update`
  - Update the device shadow.

**Note**

When you are done working with the device shadow, call `SHADOW_ClientDisconnect` to disconnect the shadow client and free system resources.

### Amazon FreeRTOS Transport Layer Security (TLS)

The Amazon FreeRTOS Transport Layer Security (TLS) interface is a thin, optional wrapper used to abstract cryptographic implementation details away from the Secure Sockets interface above it in the protocol stack. The purpose of the TLS interface is to make the current software crypto library, mbed TLS, easy to replace with an alternative implementation for TLS protocol negotiation and cryptographic primitives. The TLS interface can be swapped out without any changes required to the Secure Sockets interface. See `aws_tls.h` in the Amazon FreeRTOS source code repository.

The TLS interface is optional because you can choose to interface directly from Secure Sockets into a crypto library. The interface is not used for MCU solutions that include a full-stack offload implementation of TLS and network transport.

### Amazon FreeRTOS Wi-Fi Library

**Overview**

The Amazon FreeRTOS Wi-Fi library abstracts port-specific Wi-Fi implementations into a common API that simplifies application development and porting for all Amazon FreeRTOS-qualified boards with Wi-Fi capabilities. Using this common API, applications can communicate with their lower-level wireless stack through a common interface.

The source files for the Amazon FreeRTOS Wi-Fi library are located in `AmazonFreeRTOS/lib/wifi/portable`.
Dependencies and Requirements

The Amazon FreeRTOS Wi-Fi library requires the FreeRTOS+TCP core.

Features

The Wi-Fi library includes the following features:

- Support for WEP, WPA, and WPA2 authentication
- Access Point Scanning
- Power management
- Network profiling

For more information about the features of the Wi-Fi library, see below.

Wi-Fi Modes

Wi-Fi devices can be in one of three modes: Station, Access Point, or P2P. You can get the current mode of a Wi-Fi device by calling WIFI_GetMode. You can set a device's wi-fi mode by calling WIFI_SetMode. Switching modes by calling WIFI_SetMode disconnects the device, if it is already connected to a network.

Station mode

Set your device to Station mode to connect the board to an existing access point.

Access Point (AP) mode

Set your device to AP mode to make the device an access point for other devices to connect to. When your device is in AP mode, you can connect another device to your FreeRTOS device and configure the new Wi-Fi credentials. To configure AP mode, call WIFI_ConfigureAP. To put your device into AP mode, call WIFI_StartAP. To turn off AP mode, call WIFI_StopAP.

P2P mode

Set your device to P2P mode to allow multiple devices to connect to each other directly, without an access point.

Security

The Wi-Fi API supports WEP, WPA, and WPA2 security types. When a device is in Station mode, you must specify the network security type when calling the WIFI_ConnectAP function. When a device is in AP mode, the device can be configured to use any of the supported security types:

- eWiFiSecurityOpen
- eWiFiSecurityWEP
- eWiFiSecurityWPA
- eWiFiSecurityWPA2

Scanning and Connecting

To scan for nearby access points, set your device to Station mode, and call the WIFI_Scan function. If you find a desired network in the scan, you can connect to the network by calling WIFI_ConnectAP and providing the network credentials. You can disconnect a Wi-Fi device from the network by calling
**Power Management**

Different Wi-Fi devices have different power requirements, depending on the application and available power sources. A device might always be powered on to reduce latency or it might be intermittently connected and switch into a low power mode when Wi-Fi is not required. The interface API supports various power management modes like always on, low power, and normal mode. You set the power mode for a device using the `WIFI_SetPMMode` function. You can get the current power mode of a device by calling the `WIFI_GetPMMode` function.

**Network Profiles**

The Wi-Fi library enables you to save network profiles in the non-volatile memory of your devices. This allows you to save network settings so they can be retrieved when a device reconnects to a Wi-Fi network, removing the need to provision devices again after they have been connected to a network. `WIFI_NetworkAdd` adds a network profile. `WIFI_NetworkGet` retrieves a network profile. `WIFI_NetworkDel` deletes a network profile. The number of profiles you can save depends on the platform.

**Footprint**

**Code Size (example generated with GCC for ARM Cortex-M)**

<table>
<thead>
<tr>
<th>File name</th>
<th>Size (with -O1 Optimization)</th>
<th>Size (with Os Optimization)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wi-Fi library, with all options enabled</td>
<td>Varies by port</td>
<td>Varies by port</td>
</tr>
<tr>
<td>For example, for the TI CC3220SF: lib/wifi/portable/ti/cc3220_launchpad/aws_wifi.c</td>
<td>3.7 K</td>
<td>3.0 K</td>
</tr>
</tbody>
</table>

**Source and Header Files**

```
Amazon FreeRTOS
|-- lib
  |-- include
  |   |-- aws_wifi.h         [Include to use the AFR WIFI API]
  |   |-- wifi
  |     |-- portable
  |     |   |-- ...              [Port-specific folder structure]
  |     |     |-- aws_wifi.c     [Required to use the AFR WIFI API]
```

**Configuration**

To use the Wi-Fi library, you need to define several identifiers in a configuration file. For information about these identifiers, see the API Reference (p. 106).
Note
The library does not include the required configuration file. You must create one. When creating your configuration file, be sure to include any board-specific configuration identifiers that your board requires.

Initialization

Before you use the Wi-Fi library, you need to initialize some board-specific components, in addition to the FreeRTOS components. Using the demos/vendor/board/common/application_code/main.c file as a template for initialization, do the following:

1. Remove the sample Wi-Fi connection logic in main.c if your application handles Wi-Fi connections. Replace the following DEMO_RUNNER_RunDemos() function call:

```c
if( SYSTEM_Init() == pdPASS )
{
...
  DEMO_RUNNER_RunDemos();
...
}
```

With a call to your own application:

```c
if( SYSTEM_Init() == pdPASS )
{
...
  // This function should create any tasks
  // that your application requires to run.
  YOUR_APP_FUNCTION();
...
}
```

2. Call WIFI_On() to initialize and power on your Wi-Fi chip.

   Note
   Some boards might require additional hardware initialization.

3. Pass a configured WFINetworkParams_t structure to WIFI_ConnectAP() to connect your board to an available Wi-Fi network. For more information about the WFINetworkParams_t structure, see Example Usage (p. 106) and API Reference (p. 106).

API Reference

For a full API reference, see Wi-Fi API Reference.

Example Usage

Connecting to a Known AP

```c
#define clientcredentialWIFI_SSID    "MyNetwork"
#define clientcredentialWIFI_PASSWORD   "hunter2"

INetworkParams_t xNetworkParams;
WIFIReturnCode_t xWifiStatus;

xWifiStatus = WIFI_On(); // Turn on Wi-Fi module
// Check that Wi-Fi initialization was successful
```
Scanning for nearby APs

WIFINetworkParams_t xNetworkParams;
WIFIReturnCode_t xWifiStatus;

configPRINT(("Turning on wifi...\n"));
xWifiStatus = WIFI_On();

configPRINT(("Checking status...\n"));
if( xWifiStatus == eWiFiSuccess )
{
    configPRINT(( "WiFi module initialized.\n") );
// Handle module init failure
}
else
{
    configPRINTF( ( "WiFi module failed to initialize.\n" ) );
// Handle module init failure
}

WIFI_SetMode(eWiFiModeStation);

/* Some boards might require additional initialization steps to use the Wi-Fi library. */

while (1){
    configPRINT("Starting scan\n");
    const uint8_t ucNumNetworks = 12; //Get 12 scan results
    WIFIScanResult_t xScanResults[ ucNumNetworks ];
    xWifiStatus = WIFI_Scan( xScanResults, ucNumNetworks ); // Initiate scan
    configPRINT("Scan started\n");

    // For each scan result, print out the SSID and RSSI
    if ( xWifiStatus == eWiFiSuccess ){
Porting

The aws_wifi.c implementation needs to implement the functions defined in aws_wifi.h. At the very least, the implementation needs to return eWiFiNotSupported for any non-essential or unsupported functions.

Amazon FreeRTOS Over-the-Air Updates

Over-the-air (OTA) updates allow you to deploy files to one or more devices in your fleet. Although OTA updates were designed to be used to update device firmware, you can use them to send any files to one or more devices registered with AWS IoT. When you send files over the air, it is a best practice to digitally sign them so that the devices that receive the files can verify they have not been tampered with en route. You can use Code Signing for Amazon FreeRTOS to sign and encrypt your files or you can sign your files with your own code-signing tools.

When you create an OTA update, the OTA Update Manager Service (p. 143) creates an AWS IoT job to notify your devices that an update is available. The OTA demo application runs on your device and creates an Amazon FreeRTOS task that subscribes to notification topics for AWS IoT jobs and listens for update messages. When an update is available, the OTA agent publishes requests to AWS IoT streaming topics and receives file blocks using the MQTT protocol. It reassembles the blocks into files and checks the digital signature of the downloaded files. If the files are valid, it installs the firmware update. If you are not using the Amazon FreeRTOS OTA Update demo application, you must integrate the Amazon FreeRTOS Over-the-Air (OTA) Agent Library (p. 93) into your own application to get the firmware update capability.

Amazon FreeRTOS over-the-air updates make it possible for you to:

- Digitally sign and encrypt firmware before deployment.
- Deploy new firmware images to a single device, a group of devices, or your entire fleet.
- Deploy firmware to devices as they are added to groups, reset, or reprovisioned.
- Verify the authenticity and integrity of new firmware after it's deployed to devices.
- Monitor the progress of a deployment.
- Debug a failed deployment.

Over-the-Air Update Prerequisites

To use over-the-air updates, you need to:

- Create an S3 bucket to store your firmware update.
- Create an OTA service role.
- Create an OTA user policy.
- Create or purchase a code-signing certificate.
• If you are using Code Signing for Amazon FreeRTOS, import your code-signing key into ACM.
• If you are using Code Signing for Amazon FreeRTOS, create a code-signing policy.
• Download Amazon FreeRTOS with the OTA library for your platform or, if you are not using Amazon FreeRTOS, provide your own OTA agent implementation.

Create an Amazon S3 Bucket to Store Your Update

OTA update files are stored in Amazon S3 buckets. If you are using Code Signing for Amazon FreeRTOS, the command you use to create a code-signing job takes a source bucket (where the unsigned firmware image is located) and a destination bucket (where the signed firmware image is written). You can specify the same bucket for both the source and destination. The file names are changed to GUIDs so the original files are not overwritten.

To create an Amazon S3 bucket

2. Choose Create bucket.
3. Type a bucket name, and then choose Next.
   Note
   Your bucket name must begin with afr-ota.
4. On the Create bucket page, choose Versioning.
5. Choose Enable versioning, choose Save, and then choose Next.
6. Choose Next to accept the default permissions.
7. Choose Create bucket.

For more information about Amazon S3, see Amazon Simple Storage Service Console User Guide.

Creating an OTA Update Service Role

The OTA Update service assumes this role to create and manage OTA update jobs on your behalf.

To create an OTA service role

2. From the navigation pane, choose Roles.
3. Choose Create role.
4. Under Select type of trusted entity, choose AWS Service.
5. Choose IoT from the list of AWS services.
6. Under Select your use case, choose IoT allows IoT to call AWS services on your behalf.
7. Choose Next: Permissions.
8. Choose Next: Review.
9. Type a role name and description, and then choose Create role.

For more information about IAM roles, see IAM Roles.

To add OTA update permissions to your OTA service role

1. In the search box on the IAM console page, enter the name of your role, and then choose it from the list.
2. Choose Attach policy.
3. In the Search box, enter AmazonFreeRTOSOTAUpdate. In the list of managed policies, select AmazonFreeRTOSOTAUpdate, and then choose Attach policy.

**To add the required permissions to your OTA service role**

1. In the search box on the IAM console page, enter the name of your role and then choose it from the list.
2. In the lower right, choose Add inline policy.
3. Choose the JSON tab.
4. Copy and paste the following policy document into the text box. Replace \(<example-bucket>\) with the name of your bucket.

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": "iam:PassRole",
         "Resource":
            "arn:aws:iam::<your_account_id>:role/<your_role_name>"
      }
   ]
}
```

If you provide your own bucket name, use the following policy to grant your service role access to your bucket:

```json
"Version": "2012-10-17",
"Statement": [
   {
      "Effect": "Allow",
      "Action": [
         "s3:GetObjectVersion",
         "s3:GetObject"
      ],
      "Resource": "arn:aws:s3::<example-bucket>/*"
   }
]
```

5. Choose Review policy.
6. Enter a name for the policy and then choose Create policy.

**Creating an OTA User Policy**

You must grant your IAM user permission to perform over-the-air updates. Your IAM user must have permissions to:

- Access the S3 bucket where your firmware updates are stored.
- Access certificates stored in AWS Certificate Manager.
- Access the AWS IoT Streaming service.
- Access Amazon FreeRTOS OTA updates.
- Access AWS IoT jobs.
- Access IAM.
- Access Code Signing for Amazon FreeRTOS.
• List Amazon FreeRTOS hardware platforms.

To grant your IAM user the required permissions, create an OTA user policy and then attach it to your IAM user. For more information, see IAM Policies.

To create an OTA user policy

2. In the navigation pane, choose Users.
3. Choose your IAM user from the list.
4. Choose Add permissions.
5. Choose Attach existing policies directly.
6. Choose Create policy.
7. Choose the JSON tab, and copy and paste the following policy document into the policy editor:

```json
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect":"Allow",
      "Action": [
        "s3:ListBucket",
        "s3:ListAllMyBuckets",
        "s3:CreateBucket",
        "s3:PutBucketVersioning",
        "s3:GetBucketLocation",
        "s3:GetObjectVersion",
        "acm:ImportCertificate",
        "acm:ListCertificates",
        "iot:*",
        "iam:ListRoles",
        "freertos:ListHardwarePlatforms",
        "freertos:DescribeHardwarePlatform"
      ],
      "Resource": "*"
    },
    {
      "Effect":"Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::<example-bucket>/*"
    },
    {
      "Effect":"Allow",
      "Action": "iam:PassRole",
      "Resource": "arn:aws:iam::<your-account-id>::role/<role-name>"
    }
  ]
}
```

Replace `<example-bucket>` with the name of the Amazon S3 bucket where your OTA update firmware image is stored. Replace `<your-account-id>` with your AWS account ID. You can find your AWS account ID in the upper right of the console. When you enter your account ID, remove any dashes (-). Replace `<role-name>` with the name of the IAM service role you just created.

9. Enter a name for your new OTA user policy, and then choose Create policy.
To attach the OTA user policy to your IAM user

1. In the IAM console, in the navigation pane, choose **Users**, and then choose your user.
2. Choose **Add permissions**.
3. Choose **Attach existing policies directly**.
4. Search for the OTA user policy you just created and select the check box next to it.
5. Choose **Next: Review**.
6. Choose **Add permissions**.

Creating a Code-Signing Certificate

To digitally sign firmware images, you need a code-signing certificate and private key. For testing purposes, you can create a self-signed certificate and private key. For production environments, purchase a certificate through a well-known certificate authority (CA).

Different platforms require different types of code-signing certificates. The following section describes how to create code-signing certificates for each of the Amazon FreeRTOS-qualified platforms.

Creating a Code-Signing Certificate for the Texas Instruments CC3200SF-LAUNCHXL

The SimpleLink Wi-Fi CC3220SF Wireless Microcontroller Launchpad Development Kit supports two certificate chains for firmware code signing:

- Production (certificate-catalog)

  To use the production certificate chain, you must purchase a commercial code-signing certificate and use the **TI Uniflash tool** to set the board to production mode.

- Testing and development (certificate-playground)

  The playground certificate chain allows you to try out OTA updates with a self-signed code-signing certificate.

Install the **SimpleLink CC3220 SDK version 2.10.00.04**. By default, the files you need are located here:

C:\ti\simplelink_cc32xx_sdk_2_10_00_04\tools\cc32xx_tools\certificate-playground (Windows)

/Applications/Ti/simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground (macOS)

The certificates in the SimpleLink CC3220 SDK are in DER format. To create a self-signed code-signing certificate, you must convert them to PEM format.

Follow these steps to create a code-signing certificate that is linked to the Texas Instruments playground certificate hierarchy and meets AWS Certificate Manager and Code Signing for Amazon FreeRTOS criteria.

To create a self-signed code signing certificate

1. In your working directory, use the following text to create a file named **cert_config**. Replace **test_signer@amazon.com** with your email address.

```
[ req ]
```

prompt = no
distinguished_name = my dn

[ my dn ]
commonName = test_signer@amazon.com

[ my_exts ]
keyUsage = digitalSignature
extendedKeyUsage = codeSigning

2. Create a private key and certificate signing request (CSR):

openssl req -config cert_config -extensions my_exts -nodes -days 365 -newkey rsa:2048 -keyout tisigner.key -out tisigner.csr

3. Convert the Texas Instruments playground root CA private key from DER format to PEM format.
The TI playground root CA private key is located here:

C:\ti\simplelink_cc32xx_sdk_2_10_00_04\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert-key (Windows)
/Applications/Ti/simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert-key (macOS)

openssl rsa -inform DER -in dummy-root-ca-cert-key -out dummy-root-ca-cert-key.pem

4. Convert the Texas Instruments playground root CA certificate from DER format to PEM format.
The TI playground root certificate is located here:

C:\ti\simplelink_cc32xx_sdk_2_10_00_04\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert (Windows)
/Applications/Ti/simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert (macOS)

openssl x509 -inform DER -in dummy-root-ca-cert -out dummy-root-ca-cert.pem

5. Sign the CSR with the Texas Instruments root CA:

openssl x509 -extfile cert_config -extensions my_exts -req -days 365 -in tisigner.csr -CA dummy-root-ca-cert.pem -CAkey dummy-root-ca-cert-key.pem -set_serial 01 -out tisigner.crt.pem -sha1

6. Convert your code-signing certificate (tisigner.crt.pem) to DER format:

openssl x509 -in tisigner.crt.pem -out tisigner.crt.der -outform DER

Note
You write the tisigner.crt.der certificate onto the TI development board later.

7. Import the code-signing certificate, private key, and certificate chain into AWS Certificate Manager:


This command displays an ARN for your certificate. You need this ARN when you create an OTA update job.
Note
This step is written with the assumption that you are going to use Code Signing for Amazon FreeRTOS to sign your firmware images. Although the use of Code Signing for Amazon FreeRTOS is recommended, you can sign your firmware images manually.

Creating a Code-Signing Certificate for the Microchip Curiosity PIC32MZEF

The Microchip Curiosity PIC32MZEF supports a self-signed SHA256 with ECDSA code-signing certificate.

1. In your working directory, use the following text to create a file named cert_config. Replace test_signer@amazon.com with your email address:

   ```
   [ req ]
   prompt = no
   distinguished_name = my_dn
   
   [ my_dn ]
   commonName = test_signer@amazon.com
   
   [ my_exts ]
   keyUsage = digitalSignature
   extendedKeyUsage = codeSigning
   ```

2. Create an ECDSA code-signing private key:

   ```
   openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
   ```

3. Create an ECDSA code-signing certificate:

   ```
   openssl req -new -x509 -config cert_config -extensions my_exts -nodes -days 365 -key ecdsasigner.key -out ecdsasigner.crt
   ```

4. Import the code-signing certificate, private key, and certificate chain into AWS Certificate Manager:

   ```
   aws acm import-certificate --certificate file://ecdsasigner.crt --private-key file://ecdsasigner.key
   ```

   This command displays an ARN for your certificate. You need this ARN when you create an OTA update job.

   Note
   This step is written with the assumption that you are going to use Code Signing for Amazon FreeRTOS to sign your firmware images. Although the use of Code Signing for Amazon FreeRTOS is recommended, you can sign your firmware images manually.

Creating a Code-Signing Certificate for the Espressif ESP32

The Espressif ESP32 boards support a self-signed SHA256 with ECDSA code-signing certificate.

1. In your working directory, use the following text to create a file named cert_config. Replace test_signer@amazon.com with your email address:

   ```
   [ req ]
   prompt = no
   distinguished_name = my_dn
   ```
2. Create an ECDSA code-signing private key:

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. Create an ECDSA code-signing certificate:

```
openssl req -new -x509 -config cert_config -extensions my_exts -nodes -days 365 -key
edcdasigner.key -out ecdsasigner.crt
```

4. Import the code-signing certificate, private key, and certificate chain into AWS Certificate Manager:

```
aws acm import-certificate --certificate file://ecdsasigner.crt --private-key
file://ecdsasigner.key
```

This command displays an ARN for your certificate. You need this ARN when you create an OTA update job.

**Note**

This step is written with the assumption that you are going to use Code Signing for Amazon FreeRTOS to sign your firmware images. Although the use of Code Signing for Amazon FreeRTOS is recommended, you can sign your firmware images manually.

### Creating a Code-Signing Certificate for the Amazon FreeRTOS Windows Simulator

The Amazon FreeRTOS Windows simulator requires a code-signing certificate with an ECDSA P-256 key and SHA-256 hash to perform OTA updates. If you don’t have a code-signing certificate, use these steps to create one:

1. In your working directory, use the following text to create a file named `cert_config`. Replace `test_signer@amazon.com` with your email address:

```
[ req ]
prompt             = no
distinguished_name = my_dn

[ my_dn ]
commonName = test_signer@amazon.com

[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

2. Create an ECDSA code-signing private key:

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. Create an ECDSA code-signing certificate:
openssl req -new -x509 -config cert_config -extensions my_exts -nodes -days 365 -key ecdssigner.key -out ecdssigner.crt

4. Import the code-signing certificate, private key, and certificate chain into AWS Certificate Manager:

    aws acm import-certificate --certificate file://ecdsasigner.crt --private-key file://ecdsasigner.key

This command displays an ARN for your certificate. You need this ARN when you create an OTA update job.

**Note**

This step is written with the assumption that you are going to use Code Signing for Amazon FreeRTOS to sign your firmware images. Although the use of Code Signing for Amazon FreeRTOS is recommended, you can sign your firmware images manually.

**Creating a Code-Signing Certificate for Custom Hardware**

Using an appropriate toolset, create a self-signed certificate and private key for your hardware.

After you create your code-signing certificate, import it into ACM:

    aws acm import-certificate --certificate file://code-sign.crt --private-key file://code-sign.key

The output from this command displays an ARN for your certificate. You need this ARN when you create an OTA update job.

ACM requires certificates to use specific algorithms and key sizes. For more information, see Prerequisites for Importing Certificates. For more information about ACM, see Importing Certificates into AWS Certificate Manager.

You must copy, paste, and format the contents of your code-signing certificate and private key into the aws_ota_codesigner_certificate.h file that is part of the Amazon FreeRTOS code you download later.

**Granting Access to Code Signing for Amazon FreeRTOS**

In production environments, you should digitally sign your firmware update to ensure the authenticity and integrity of the update. You can sign your update manually or you can use Code Signing for Amazon FreeRTOS to sign your code. To use Code Signing for Amazon FreeRTOS, you must grant your IAM user account access to Code Signing for Amazon FreeRTOS.

**To grant your IAM user account permissions for Code Signing for Amazon FreeRTOS**

2. In the navigation pane, choose Policies.
3. Choose Create Policy.
4. On the JSON tab, copy and paste the following JSON document into the policy editor. This policy allows the IAM user access to all code-signing operations.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```
5. Choose **Review policy**.
6. Enter a policy name and description, and then choose **Create policy**.
7. In the navigation pane, choose **Users**.
8. Choose your IAM user account.
9. On the **Permissions** tab, choose **Add permissions**.
10. Choose **Attach existing policies directly**, and then select the check box next to the code-signing policy you just created.
11. Choose **Next: Review**.
12. Choose **Add permissions**.

## Download Amazon FreeRTOS with the OTA Library

Follow the steps in this section to download code and build demo applications.

### Download and Build Amazon FreeRTOS for the Texas Instruments CC3200SF-LAUNCHXL

**To download Amazon FreeRTOS and the OTA demo code**

1. Browse to the AWS IoT console and from the navigation pane, choose **Software**.
2. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
3. From the list of software configurations, choose **Connect to AWS IoT - TI**. Choose the configuration name, not the **Download** link.
4. Under **Libraries**, choose **Add another library**, and then choose **OTA Updates**.
5. Under **Demo Projects**, choose **OTA Updates**.
6. Under **Name required**, enter **Connect-to-IoT-OTA-TI**, and then choose **Create and download**.

Save the zip file that contains Amazon FreeRTOS and the OTA demo code to your computer.

**To build the demo application**

1. Extract the .zip file.
2. Follow the instructions in *Getting Started with Amazon FreeRTOS* (p. 4), to import the aws_demos project into Code Composer Studio, configure your AWS IoT endpoint, your Wi-Fi SSID and password, and a private key and certificate for your board.
3. Open the project in Code Composer Studio and open **demos/common/demo_runner/aws_demo_runner.c**. Find the **DEMO_RUNNER_RunDemos** function and make sure all function calls are commented out except **vStartOTAUpdateDemoTask**.
4. Build the solution and make sure it builds without errors.
5. Start a terminal emulator and use the following settings to connect to your board:

- Baud rate: 115200
- Data bits: 8
Amazon FreeRTOS User Guide
Over-the-Air Update Prerequisites

• Parity: None
• Stop bits: 1

6. Run the project on your board to make sure it can connect to Wi-Fi and the AWS IoT MQTT message broker.

When run, the terminal emulator should display text like the following:

```
0 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created
Device came up in Station mode
2 142 [Tmr Svc] Wi-Fi module initialized.
3 142 [Tmr Svc] Starting key provisioning...
4 142 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 340 [Tmr Svc] Write device certificate...
7 433 [Tmr Svc] Key provisioning done...
[NETAPP EVENT] IP acquired by the device
Device has connected to Mobile
Device IP Address is 192.168.111.12
8 2666 [Tmr Svc] Wi-Fi connected to AP Mobile.
9 2666 [Tmr Svc] IP Address acquired 192.168.111.12
10 2667 [OTA] OTA demo version 0.9.2
11 2667 [OTA] Creating MQTT Client...
12 2667 [OTA] Connecting to broker...
13 3512 [OTA] Connected to broker.
14 3715 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/OtaGA/jobs/$next/get/accepted
15 4018 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/OtaGA/jobs/notif-accept
16 4027 [OTA Task] [prvPAL_GetPlatformImageState] xFileInfo.Flags = 0250
17 4027 [OTA Task] [prvPAL_GetPlatformImageState] eOTA_PAL_ImageState_Valid
18 4034 [OTA Task] [OTA_CheckForUpdate] Request #0
19 4248 [OTA] [OTA_AgentInit] Ready.
20 4249 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken: 0:OtaGA ]
21 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
22 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
23 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
24 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: afr_ota
25 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
26 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: files
27 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
28 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
29 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
30 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
31 4251 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-shal-rsa
32 4251 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
33 4251 [OTA Task] [prvOTA_Close] Context->0x2001b2c4
34 5248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
35 6248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
36 7248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
37 8248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
38 9248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
```

Download and Build Amazon FreeRTOS for the Microchip Curiosity PIC32MZEF

To download the Amazon FreeRTOS OTA demo code

1. Browse to the AWS IoT console and from the navigation pane, choose Software.
2. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
3. From the list of software configurations, choose **Connect to AWS IoT - Microchip**. Choose the configuration name, not the **Download** link.
4. Under **Libraries**, choose **Add another library**, and then choose **OTA Updates**.
5. Under **Demo projects**, choose **OTA Update**.
6. Under **Name required**, enter a name for your custom Amazon FreeRTOS software configuration.
7. Choose **Create and download**.

**To build the OTA update demo application**

1. Extract the .zip file you just downloaded.
2. Follow the instructions in **Getting Started with Amazon FreeRTOS (p. 4)** to import the aws_demos project into the MPLAB X IDE, configure your AWS IoT endpoint, your Wi-Fi SSID and password, and a private key and certificate for your board.
3. Open **aws_demos/lib/aws/ota/aws_ota_codesigner_certificate.h**.
4. Paste the contents of your code-signing certificate into the static const char signingcredentialSIGNING_CREDENTIAL_PEM variable. Following the same format as aws_clientcredential_keys.h, each line must end with the new line character ('\n') and be enclosed in quotation marks.

For example, your certificate should look similar to the following:

```
"-----BEGIN CERTIFICATE-----
"MIIBXTCCAQOgAwIBAgIJAAM4DeybZ2CTwRMMAoGCCqGSM49BAMCMCEExHbAdBgNV
"BAMcMMoGCCqGSM49BAMCMAoGCSqGSIb3DQEBCwUAA4GAIIBAIBzceZjOAcQ7kOGh
"DBYg9z:fWv2vVL1X+E4dIF7dbkVNUh4IrjJ1CAzFkcs8
"gXnZn83H40XMK1tD5dFwz9ng7s8+QyQg?ygmr2stz5yoh60XMC1wCWFfDPQ
"BAD0Xe6MNGA1UdJQQMQMaoGCCsGAgUFwMDMaoGCCqGSM49BAMCA0gAMEUCIF0P
"r5cb7rEUNT0Wo4d5MacrgOABfsOYVb0K9fP63WAqt5h3BaS123cKSGg84twlq
"Tk/vp/xEmyzMz2dV+HxV/OM="\n"-----END CERTIFICATE-----"
```
5. Install **Python 3** or later.
6. Install pyOpenSSL by running `pip install pyopenssl`.
7. Copy your code-signing certificate in .pem format in the path `\demos\common\ota \bootloader\utility\codesigner_cert_utility\`. Rename the certificate file aws_ota_codesigner_certificate.pem.
8. Open the project in MPLAB X IDE and open `demos/common/demo_runner/aws_demo_runner.c`. Find the DEMO_RUNNER_RunDemos function and make sure all function calls are commented out except vStartOTAUpdateDemoTask.
9. Build the solution and make sure it builds without errors.
10. Start a terminal emulator and use the following settings to connect to your board:
    - Baud rate: 115200
    - Data bits: 8
    - Parity: None
    - Stop bits: 1
11. Unplug the debugger from your board and run the project on your board to make sure it can connect to Wi-Fi and the AWS IoT MQTT message broker.

When you run the project, the MPLAB X IDE should open an output window. Make sure the ICD4 tab is selected. You should see the following output.
Over-the-Air Update Prerequisites

Bootloader version 00.09.00
[prvBOOT_Init] Watchdog timer initialized.
[prvBOOT_Init] Crypto initialized.

[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] No application image or magic code present at: 0xbd000000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd000000

[prvValidateImage] Validating image at Bank : 1
[prvValidateImage] No application image or magic code present at: 0xbd100000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd100000

[prvBOOT_ValidateImages] Booting default image.

>0 36246 [IP-task] vDHCPProcess: offer ac140a0eip
  1 36297 [IP-task] vDHCPProcess: offer
  2 36297 [IP-task]
IP Address: 172.20.10.14
3 36297 [IP-task] Subnet Mask: 255.255.255.240
4 36297 [IP-task] Gateway Address: 172.20.10.1
5 36297 [IP-task] DNS Server Address: 172.20.10.1

6 36299 [OTA] OTA demo version 0.9.2
7 36299 [OTA] Creating MQTT Client...
8 36299 [OTA] Connecting to broker...
9 38673 [OTA] Connected to broker.
10 38793 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/devthingota/jobs/
    #next/get/accepted
11 38863 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/devthingota/jobs/
    notify-next
12 38863 [OTA Task] [OTA_CheckForUpdate] Request #0
13 38964 [OTA] [OTA_AgentInit] Ready.
14 38973 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken: 0:devthingota]
15 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
16 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
17 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
18 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
19 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
20 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
21 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
22 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha256-ecdsa
23 38975 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
24 38975 [OTA Task] [prvOTA_Close] Context-->0x8003b620
25 38975 [OTA Task] [prvOTA_Close] Context-->0x8003b620
26 38975 [OTA Task] [prvOTA_Close] Context-->0x8003b620
27 38975 [OTA Task] [prvPAL_Abort] Abort - OK
28 39964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
29 41964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
30 41964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
31 42964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
32 43964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
33 44964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
34 45964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
35 46964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
36 47964 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0

The terminal emulator should display text like the following:

AWS Validate: no valid signature in descr: 0xbd000000
AWS Validate: no valid signature in descr: 0xbd100000

> AWS Launch: No Map performed. Running directly from address: 0x9d000020?
AWS Launch: wait for app at: 0x9d000020
WILC1000: Initializing...
0 0

>[None] Seed for randomizer: 1172751941
1 0 [None] Random numbers: 00004272 00003B34 00000602 00002DE3
Chip ID 1503a0

[spi_cmd_rsp] [356] [nmi spi]: Failed cmd response read, bus error...

[spi_read_reg] [1086] [nmi spi]: Failed cmd response, read reg (0000108c)...

[spi_read_reg] [1116] Reset and retry 10 108c

Firmware ver. : 4.2.1
Min driver ver : 4.2.1
Curr driver ver: 4.2.1

WILC1000: Initialization successful!

Start Wi-Fi Connection...

Wi-Fi Connected
2 7219 [IP-task] DHCPProcess: offer c0a804beip
3 7230 [IP-task] DHCPProcess: offer c0a804beip
4 7230 [IP-task]

IP Address: 192.168.4.190
5 7230 [IP-task] Subnet Mask: 255.255.240.0
6 7230 [IP-task] Gateway Address: 192.168.0.1
7 7230 [IP-task] DNS Server Address: 208.67.222.222

8 7232 [OTA] OTA demo version 0.9.0
9 7232 [OTA] Creating MQTT Client...
10 7232 [OTA] Connecting to broker...
11 7232 [OTA] Sending command to MQTT task.
12 7232 [MQTT] Received message 10000 from queue.
13 8501 [IP-task] Socket sending wakeup to MQTT task.
14 10207 [MQTT] Received message 0 from queue.
15 10256 [IP-task] Socket sending wakeup to MQTT task.
16 10256 [MQTT] Received message 0 from queue.
17 10256 [MQTT] MQTT Connect was accepted. Connection established.
18 10256 [MQTT] Notifying task.
19 10257 [OTA] Command sent to MQTT task passed.
20 10257 [OTA] Connected to broker.
21 10258 [OTA Task] Sending command to MQTT task.
22 10258 [MQTT] Received message 20000 from queue.
23 10306 [IP-task] Socket sending wakeup to MQTT task.
24 10306 [MQTT] Received message 0 from queue.
25 10306 [MQTT] MQTT Subscribe was accepted. Subscribed.
26 10306 [MQTT] Notifying task.
27 10307 [OTA Task] Command sent to MQTT task passed.
28 10307 [OTA Task] [OTA] Subscribed to topic: #aws/things/Microchip/jobs/#next/get/accepted

29 10307 [OTA Task] Sending command to MQTT task.
30 10307 [MQTT] Received message 30000 from queue.
31 10336 [IP-task] Socket sending wakeup to MQTT task.
32 10336 [MQTT] Received message 0 from queue.
33 10336 [MQTT] MQTT Subscribe was accepted. Subscribed.
This output shows the Microchip Curiosity PIC32MZEF is able to connect to AWS IoT and subscribe to the MQTT topics required for OTA updates. The `Missing job parameter` messages are expected because there are no OTA update jobs pending.

Download and Build Amazon FreeRTOS for the Espressif ESP32

1. Download the Amazon FreeRTOS source from GitHub. Create a project in your IDE that includes all required sources and libraries.
2. Follow the instructions in Getting Started with Espressif to set up the required GCC-based toolchain.
3. Open `demos/common/demo_runner/aws_demo_runner.c` in a text editor. Find the `DEMO_RUNNER_RunDemos()` function and make sure all function calls are commented out except `vStartOTAUpdateDemoTask`.
4. Build the demo project by running `make` in the `demos/espressif/esp32_devkitc_esp_wrover_kit/make/` directory. You can flash the demo program and verify its output by running `make flash monitor`, as described in Getting Started with Espressif.
5. Before running the OTA Update demo:
   - Make sure that `vStartOTAUpdateDemoTask` is the only function called in the `DEMO_RUNNER_RunDemos()` function in `demos/common/demo_runner/aws_demo_runner.c`.
   - Make sure that your SHA-256/ECDSA code-signing certificate is copied into the `demos/common/include/aws_ota_codesigner_certificate.h`. 
Download and Build Amazon FreeRTOS for a Custom Hardware Platform

Download the Amazon FreeRTOS source from GitHub. Create a project in your IDE that includes all required sources and libraries.

Build and run the project to make sure it can connect to AWS IoT.

For more information about porting Amazon FreeRTOS to a new platform, see Amazon FreeRTOS Porting Guide (p. 178).

OTA Tutorial

This section contains a tutorial for updating firmware on devices running Amazon FreeRTOS using OTA updates. You can, however, use OTA updates to send files to any devices connected to AWS IoT.

You can use the AWS IoT console or the AWS CLI to create an OTA update. The console is the easiest way to get started with OTA because it does a lot of the work for you. The AWS CLI is useful when you are automating OTA update jobs, working with a large number of devices, or are using devices that have not been qualified for Amazon FreeRTOS. For more information about qualifying devices for Amazon FreeRTOS, see the Amazon FreeRTOS Partners website.

To create a OTA update
1. Deploy an initial version of your firmware to one or more devices.
2. Verify that the firmware is running correctly.
3. When a firmware update is required, make the code changes and build the new image.
4. If you are manually signing your firmware, sign and then upload the signed firmware image to your Amazon S3 bucket.
   
   If you are using Code Signing for Amazon FreeRTOS, upload your unsigned firmware image to an Amazon S3 bucket.
5. Create an OTA update.

The Amazon FreeRTOS OTA agent on the device receives the updated firmware image and verifies the digital signature, checksum, and version number of the new image. If the firmware update is verified, the device is reset and, based on application-defined logic, commits the update. If your devices are not running Amazon FreeRTOS, you must implement an OTA agent that runs on your devices.

Installing the Initial Firmware

To update firmware, you must install an initial version of the firmware that uses the OTA agent library to listen for OTA update jobs. If you are not running Amazon FreeRTOS, skip this step. You must copy your OTA agent implementation onto your devices instead.

Topics
- Install the Initial Version of Firmware on the Texas Instruments CC3200SF-LAUNCHXL (p. 124)
- Install the Initial Version of Firmware on the Microchip Curiosity PIC32MZEF (p. 126)
- Install the Initial Version of Firmware on the Espressif ESP32 (p. 128)
- Initial Firmware on the Windows Simulator (p. 130)
- Install the Initial Version of Firmware on a Custom Board (p. 130)
Install the Initial Version of Firmware on the Texas Instruments CC3200SF-LAUNCHXL

These steps are written with the assumption that you have already built the aws_demos project, as described in Download and Build Amazon FreeRTOS for the Texas Instruments CC3200SF-LAUNCHXL (p. 117).

1. On your Texas Instruments CC3200SF-LAUNCHXL, place the SOP jumper on the middle set of pins (position = 1) and reset the board.
2. Download and install the TI Uniflash tool.
3. Start Uniflash and from the list of configurations, choose CC3220SF-LAUNCHXL, then choose Start Image Creator.
4. Choose New Project.
5. On the Start new project page, enter a name for your project. For Device Type, choose CC3220SF. For Device Mode, choose Develop. Choose Create Project.
6. Disconnect your terminal emulator. On the right side of the Uniflash application window, choose Connect.
7. Under Files, choose User Files.
8. In the File selector pane, choose the Add File icon.
9. Browse to the /Applications/Ti/simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground directory, select dummy-root-ca-cert, choose Open, and then choose Write.
10. In the File selector pane, choose the Add File icon.
11. Browse to the working directory where you created the code-signing certificate and private key, choose tisigner.crt.der, choose Open, and then choose Write.
12. From the Action drop-down list, choose Select MCU Image, and then choose Browse to choose the firmware image to use write to your device (aws_demos.bin). This file is located in the AmazonFreeRTOS/demos/ti/cc3200_launchpad/ccs/Debug directory. Choose Open.
   a. In the file dialog box, confirm the file name is set to mcuflashimg.bin.
   b. Select the Vendor check box.
   c. Under File Token, type 1952007250.
   d. Under Private Key File Name, choose Browse and then choose tisigner.key from the working directory where you created the code-signing certificate and private key.
   e. Under Certification File Name, choose tisigner.crt.der.
   f. Choose Write.
13. In the left pane, under Files, choose Service Pack.
14. Under Service Pack File Name, choose Browse, browse to simplelink_cc32x_sdk_2_10_00_04/tools/cc32xx_tools/servicepack-cc3x20, choose sp_3.7.0.1_2.0.0.0_2.2.0.6.bin, and then choose Open.
15. In the left pane, under Files, choose Trusted Root-Certificate Catalog.
17. Under Source File, choose Browse, choose simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground\certcatalogPlayGround20160911.lst, and then choose Open.
18. Under Signature Source File, choose Browse, choose simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground\certcatalogPlayGround20160911.lst.signed.bin, and then choose Open.
19. Choose the button to save your project.
20. Choose the button.
21. Choose **Program Image (Create and Program)**.
22. After the programming process is complete, place the SOP jumper onto the first set of pins (position = 0), reset the board, and reconnect your terminal emulator to make sure the output is the same as when you debugged the demo with Code Composer Studio. Make a note of the application version number in the terminal output. You use this version number later to verify that your firmware has been updated by an OTA update.

The terminal should display output like the following:

```
0 0 [Tmr Svc] Simple Link task created
Device came up in Station mode
1 369 [Tmr Svc] Starting key provisioning...
2 369 [Tmr Svc] Write root certificate...
3 467 [Tmr Svc] Write device private key...
4 568 [Tmr Svc] Write device certificate...
SL Disconnect...
5 664 [Tmr Svc] Key provisioning done...
Device came up in Station mode
Device disconnected from the AP on an ERROR..!!
[NETAPP EVENT] IP acquired by the device
Device has connected to Guest
Device IP Address is 111.222.3.44

6 1716 [OTA] OTA demo version 0.9.0
7 1717 [OTA] Creating MQTT Client...
8 1717 [OTA] Connecting to broker...
9 1717 [OTA] Sending command to MQTT task.
10 1717 [MQTT] Received message 10000 from queue.
11 2193 [MQTT] MQTT Connect was accepted. Connection established.
12 2193 [MQTT] Notifying task.
13 2194 [OTA] Command sent to MQTT task passed.
14 2194 [OTA] Connected to broker.
15 2196 [OTA Task] Sending command to MQTT task.
16 2196 [MQTT] Received message 20000 from queue.
17 2697 [MQTT] MQTT Subscribe was accepted. Subscribed.
18 2697 [MQTT] Notifying task.
19 2698 [OTA Task] Command sent to MQTT task passed.
20 2698 [OTA Task] [OTA] Subscribed to topic: #aws/things/TI-LaunchPad/jobs/#next/get/accepted
21 2699 [OTA Task] Sending command to MQTT task.
22 2699 [MQTT] Received message 30000 from queue.
23 2800 [MQTT] MQTT Subscribe was accepted. Subscribed.
```
Install the Initial Version of Firmware on the Microchip Curiosity PIC32MZEF

These steps are written with the assumption that you have already built the aws_demos project, as described in Download and Build Amazon FreeRTOS for the Microchip Curiosity PIC32MZEF (p. 118).

To burn the demo application onto your board

1. Rebuild the aws_demos project and make sure it compiles without errors.

2. On the tool bar, choose .

3. After the programming process is complete, disconnect the ICD 4 debugger and reset the board. Reconnect your terminal emulator to make sure the output is the same as when you debugged the demo with MPLAB X IDE.

The terminal should display output similar to the following:

```
Bootloader version 00.09.00
[prvBOOT_Init] Watchdog timer initialized.
[prvBOOT_Init] Crypto initialized.

[prvValidateImage] Validating image at Bank: 0
[prvValidateImage] No application image or magic code present at: 0xbd000000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd000000

[prvValidateImage] Validating image at Bank: 1
[prvValidateImage] No application image or magic code present at: 0xbd100000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd100000

[prvBOOT_ValidateImages] Booting default image.

>0 36246 [IP-task] vDHCPProcess: offer ac140a0eip
```
The following procedure creates a unified hex file or factory image consisting of a reference bootloader and an application with a cryptographic signature. The bootloader verifies the cryptographic signature of the application on boot and supports OTA updates.

**To build and flash a factory image**

1. Make sure you have the SRecord tools installed from Source Forge. Verify that the directory that contains the srec_cat and srec_info programs is in your system path.
2. Update the OTA sequence number and application version for the factory image.
3. Build the aws_demos project.
4. Run the factory_image_generator.py script to generate the factory image.

```
factory_image_generator.py -b mplab.production.bin -p MCHP-Curiosity-PIC32MZEF -k private_key.pem -x aws_bootloader.X.production.hex
```

This command takes the following parameters:
• mplab.production.bin: The application binary.
• MCHP-Curiosity-PIC32MZEF: The platform name.
• private_key.pem: The code-signing private key.
• aws_bootloader.X.production.hex: The bootloader hex file.

When you build the aws_demos project, the application binary image and bootloader hex file are built as part of the process. Each project under the demos/microchip/ directory contains a dist/pic32mz_ef_curiosity/production/ directory that contains these files. The generated unified hex file is named mplab.production.unified.factory.hex.

5. Use the MPLab IPE tool to program the generated hex file onto the device.
6. You can check that your factory image works by watching the board's UART output as the image is uploaded. If everything is set up correctly, you should see the image boot successfully:

```
[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] Valid magic code at: 0xbd000000
[prvValidateImage] Valid image flags: 0xfc at: 0xbd000000
[prvValidateImage] Addresses are valid.
[prvValidateImage] Crypto signature is valid.
[...]
[prvBOOT_ValidateImages] Booting image with sequence number 1 at 0xbd000000
```

7. If your certificates are incorrectly configured or if an OTA image is not properly signed, you might see messages like the following before the chip's bootloader erases the invalid update. Check that your code-signing certificates are consistent and review the previous steps carefully:

```
[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] Valid magic code at: 0xbd000000
[prvValidateImage] Valid image flags: 0xfc at: 0xbd000000
[prvValidateImage] Addresses are valid.
[prvValidateImage] Crypto signature is not valid.
[prvBOOT_ValidateImages] Validation failed for image at 0xbd000000
[BOOT_FLASH_EraseBank] Bank erased at : 0xbd000000
```

Install the Initial Version of Firmware on the Espressif ESP32

This guide is written with the assumption that you have already performed the steps in Getting Started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT and Over-the-Air Update Prerequisites. Before you attempt an OTA update, you might want to run the MQTT demo project described in Getting Started with Amazon FreeRTOS to ensure that your board and toolchain are set up correctly.

To flash an initial factory image to the board

1. In demos/common/demo_runner/aws_demo_runner.c, in the DEMO_RUNNER_RunDemos function, comment out all functions calls except vStartOTAUpdateDemoTask.
2. With the OTA Update demo selected, follow the same steps outlined in Getting Started with ESP32 to build and flash the image. If you have previously built and flashed the project, you might need to run make clean first. After you run make flash monitor, you should see something like the following. The ordering of some messages might vary, because the demo application runs multiple tasks at once:

```
I (28) boot: ESP-IDF v3.1-dev-322-gf307f41-dirty 2nd stage bootloader
```
I (28) boot: compile time 16:32:33
I (29) boot: Enabling RNG early entropy source...
I (34) boot: SPI Speed : 40MHz
I (38) boot: SPI Mode : DIO
I (42) boot: SPI Flash Size : 4MB
I (46) boot: Partition Table:
I (50) boot: ## Label Usage Type ST Offset Length
I (57) boot: 0 nvs WiFi data 01 02 00010000 00006000
I (64) boot: 1 otadata OTA data 01 00 00016000 000002000
I (72) boot: 2 phy_init RP data 01 01 00018000 00001000
I (79) boot: 3 ota_0 OTA app 00 10 00020000 00100000
I (87) boot: 4 ota_1 OTA app 00 11 00120000 00100000
I (94) boot: 5 storage Unknown data 01 82 00220000 00010000
I (102) boot: End of partition table
I (106) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x14784 (83844)
I (144) esp_image: segment 1: paddr=0x0000347ac vaddr=0x3ffb0000 size=0x023ec (9196)
I (148) esp_image: segment 2: paddr=0x000036ba0 vaddr=0x40080000 size=0x00400 (1024)
I (151) esp_image: segment 3: paddr=0x000036fa8 vaddr=0x40080400 size=0x09068 (36968)
I (175) esp_image: segment 4: paddr=0x000040018 vaddr=0x400d0018 size=0x04934 (18740)
I (337) esp_image: segment 5: paddr=0x000b19d8 vaddr=0x40089468 size=0x04934 (18740)
I (345) esp_image: segment 6: paddr=0x000b6314 vaddr=0x400c0000 size=0x00000 (0)
I (353) boot: Loaded app from partition at offset 0x20000
I (354) boot: ota rollback check done
I (356) boot: Disabling RNG early entropy source...
I (360) cpu_start: Pro cpu up.
I (363) cpu_start: Single core mode
I (368) heap_init: Initializing. RAM available for dynamic allocation:
I (375) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (381) heap_init: At 3FFC0748 len 0001F8B8 (126 KiB): DRAM
I (387) heap_init: At 3FFE04350 len 0001BCB0 (111 KiB): D/IRAM
I (400) heap_init: At 4008DD9C len 00012264 (72 KiB): IRAM
I (406) cpu_start: Pro cpu start user code
I (88) wifi_start: Starting scheduler on PRO CPU.
I (113) wifi: wifi firmware version: f79168c
I (113) wifi: config NVS flash: enabled
I (113) wifi: config nano formatting: disabled
I (113) system_api: Base MAC address is not set, read default base MAC address from BLKO of EFUSE
I (123) system_api: Base MAC address is not set, read default base MAC address from BLKO of EFUSE
I (133) wifi: Init dynamic tx buffer num: 32
I (143) wifi: Init data frame dynamic rx buffer num: 32
I (143) wifi: Init management frame dynamic rx buffer num: 32
I (143) wifi: wifi driver task: 3ffcc7ec, prio:23, stack:4096
I (153) wifi: Init static rx buffer num: 10
I (153) wifi: Init dynamic rx buffer num: 32
I (163) wifi: wifi power manager task: 0x3ffcc028 prio: 21 stack: 2560
I (233) phy: phy_version: 383.0, 79a622c, Jan 30 2018, 15:38:06, 0, 0
I (233) wifi: mode : sta (30:ae:a4:80:0a:04)
I (363) wifi: n:1 0, o:1 0, ap:255 255, sta:1 0, prof:1
I (1343) wifi: state: init -> auth (0)
I (1343) wifi: state: auth -> assoc (0)
I (1353) wifi: state: assoc -> run (10)
I (1373) wifi: connected with <Your_WiFi_SSID>, channel 1
I (1373) WIFI: SYSTEM_EVENT_STA_CONNECTED
3. The ESP32 board is now listening for OTA updates. The ESP-IDF monitor is launched by the `make flash monitor` command. You can press Ctrl+] to quit. You can also use your favorite TTY terminal program (for example, PuTTY, Tera Term, or GNU Screen) to listen to the board's serial output; examples might include. Be aware that connecting to the board's serial port might cause it to reboot.

**Initial Firmware on the Windows Simulator**

When you use the Windows simulator, there is no need to flash an initial version of the firmware. The Windows simulator is part of the `aws_demos` application, which also includes the firmware.

**Install the Initial Version of Firmware on a Custom Board**

Using your IDE, build the `aws_demos` project, making sure to include the OTA library. For more information about the structure of the Amazon FreeRTOS source code, see Navigating the Demo Applications (p. 156).

Make sure to include your code-signing certificate, private key, and certificate trust chain either in the Amazon FreeRTOS project or on your device.

Using the appropriate tool, burn the application onto your board and make sure it is running correctly.
Update the Version of Your Firmware

The OTA agent included with Amazon FreeRTOS checks the version of any update and installs it only if it is more recent than the existing firmware version. The following steps show you how to increment the firmware version of the OTA demo application.

1. Open the `aws_demos` project in your IDE.
2. Open `demos/common/include/aws_application_version.h` and increment the `APP_VERSION_BUILD` token value.
3. If you are using the Microchip Curiosity PIC32MZEF, increment the OTA sequence number in `demos\common\ota\bootloader\utility\user-config\ota-descriptor.config`. The OTA sequence number should be incremented for every new OTA image generated.
4. Rebuild the project.

You must copy your firmware update into the Amazon S3 bucket that you created as described in Create an Amazon S3 Bucket to Store Your Update (p. 109). The name of the file you need to copy to Amazon S3 depends on the hardware platform you are using:

- Texas Instruments CC3200SF-LAUNCHXL: `demos\ti\cc3220_launchpad\ccs\debug\aws_demos.bin`
- Microchip Curiosity PIC32MZEF: `demos\microchip\curiosity_pic32mzef\mplab\dist\pic32mz_ef_curiosity\production\mplab.production.ota.bin`
- Espressif ESP32: `demos\espressif\esp32_devkitc_esp_wrover_kit\make\build\aws_demos.bin`

Creating an OTA Update (AWS IoT Console)

1. In the navigation pane of the AWS IoT console, choose Manage, and then choose Jobs.
2. Choose Create.
3. Under Create an Amazon FreeRTOS Over-the-Air (OTA) update job, choose Create OTA update job.
4. You can deploy an OTA update to a single device or a group of devices. Under Select devices to update, choose Select. To update a single device, choose the Things tab. To update a group of devices, choose the Thing Groups tab.
5. If you are updating a single device, select the check box next to the IoT thing associated with your device. If you are updating a group of devices, select the check box next to the thing group associated with your devices. Choose Next.
6. Under Select and sign your firmware image, choose Sign a new firmware image for me.
8. In Create a code signing profile, enter a name for your code-signing profile.
   a. Under Device hardware platform, choose your hardware platform.
      
      \textbf{Note}  
      Only hardware platforms that have been qualified for Amazon FreeRTOS are displayed in this list. If you are using a non-qualified platform, you must use the CLI to create the OTA update. For more information, see Creating an OTA Update with the AWS CLI (p. 133).
   b. Under Code signing certificate, choose Select to select a previously imported certificate or Import to import a new certificate.
   c. Under Pathname of code signing certificate on device, enter the fully-qualified path name to the code-signing certificate on your device. This likely varies by platform.
Note
When running on the Microchip Curiosity PIC32MZEF, the code-signing certificate is first searched for in the certificate store by name. If not found, a built-in certificate is used.

Important
On the Texas Instruments CC3220SF-LAUNCHXL, do not include a leading slash character (/) in front of the file name if your code signing certificate exists in the root of the file system on this platform. Otherwise, the OTA update fails during authentication with a file not found error.

9. Under Select your firmware image in S3 or upload it, choose Select. A list of your Amazon S3 buckets is displayed. Choose the bucket that contains your firmware update, and then choose your firmware update in the bucket.

Note
The Microchip Curiosity PIC32MZEF demo projects produce two binary images with default names of mplab.production.bin and mplab.production.ota.bin. Use the second file when you upload an image for OTA updating.

10. Under Pathname of firmware image on device, enter the fully-qualified path name to the location where the firmware image will be copied onto your device. This location also varies by platform.

Important
On the Texas Instruments CC3220SF-LAUNCHXL, due to security restrictions, the firmware image path name must be /sys/mcuflashimg.bin.

11. Under IAM role for OTA update job, choose a role that allows access to your S3 bucket and has the following policies:
   • AWSIoTThingsRegistration
   • AmazonFreeRTOSOTAUpdate

12. Choose Next.
13. Under Job type, choose Your job will complete after deploying to the selected devices/groups (snapshot).
14. Enter an ID and a description for your OTA update job and then choose Create.

To use a previously signed firmware image

1. Under Select and sign your firmware image, choose Select a previously signed firmware image.
2. Under Pathname of firmware image on device, enter the fully-qualified path name to the location where the firmware image will be copied onto your device. This might be different on different platforms.
3. Under Previous code signing job, choose Select, and then choose the previous code-signing job used to sign the firmware image you are using for the OTA update.

Using a custom signed firmware image

1. Under Select and sign your firmware image, choose Use my custom signed firmware image.
2. Under Pathname of code signing certificate on device, enter the fully-qualified path name to the code-signing certificate on your device. This might be different for different platforms.
3. Under Pathname of firmware image on device, enter the fully-qualified path name to the location where the firmware image will be copied onto your device. This might be different on different platforms.
4. Under Signature, paste your PEM format signature.
5. Under Original hash algorithm, choose the hash algorithm that was used when creating your file signature.
6. Under **Original encryption algorithm**, choose the algorithm that was used when creating your file signature.

7. Under **Select your firmware image in Amazon S3**, choose the Amazon S3 bucket and the signed firmware image in the Amazon S3 bucket.

After you have specified the code-signing information, specify the OTA update job type, service role, and an ID for your update.

**Note**
Do not use any personally identifiable information in the job ID for your OTA update. Examples of personally identifiable information include:

- Your name.
- Your IP address.
- Your email address.
- Your location.
- Bank details.
- Medical information.

1. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.

2. Under **IAM role for OTA update job**, choose your OTA service role.

3. Enter an alphanumeric ID for your job and then choose **Create**.

The job appears in the AWS IoT console with a status of **IN PROGRESS**.

**Note**
The AWS IoT console does not update the state of jobs automatically. Refresh your browser to see updates.

Connect your serial UART terminal to your device. You should see output that indicates the device is downloading the updated firmware.

After the device downloads the updated firmware, it restarts and then installs the firmware. You can see what’s happening in the UART terminal.

For a complete walkthrough of how to use the console to create an OTA update, see **OTA Demo Application (p. 168)**.

**Creating an OTA Update with the AWS CLI**

To create an OTA update with the AWS CLI you:

1. Digitally sign your firmware image.

2. Create a stream of your digitally signed firmware image.


**Digitally Signing Your Firmware Update**

When you use the CLI to perform OTA updates, you can use Code Signing for Amazon FreeRTOS or sign your firmware update yourself.
Signing Your Firmware Image with Code Signing for Amazon FreeRTOS

To sign your firmware image using Code Signing for Amazon FreeRTOS, you must install the Code Signing Tools. Download the tools and read the README file for installation instructions. For more information about Code Signing for Amazon FreeRTOS, see Code Signing for Amazon FreeRTOS.

After you install and configure the code signing tools, copy your unsigned firmware image to your Amazon S3 bucket and start a code signing job with the following CLI commands. The `put-signing-profile` command creates a reusable code-signing profile. The `start-signing-job` command starts the signing job.

```bash
code
aws signer put-signing-profile --profile-name <your_profile_name>
--signing-material certificateArn=
arn:aws:acm:<your-region>:<your-aws-account-id>:certificate/<your-certificate-id>
--platform <your-hardware-platform> --signing-parameters
certname=<your_certificate_path_on_device>

aws signer start-signing-job --source
's3={bucketName=<your_s3_bucket>,key=<your_s3_object_key>,version=<your_s3_object_version_id>}'
--destination 's3={bucketName=<your_destination_bucket>}' --profile-name <your_profile_name>
```

**Note**

*<your-source-bucket-name> and <your-destination-bucket-name> can be the same Amazon S3 bucket.*

The following text describes the parameters for the `start-signing-job` command:

**source**

Specifies the location of the unsigned firmware in an S3 bucket.

- **bucketName**: The name of your S3 bucket.
- **key**: The key (file name) of your firmware in your S3 bucket.
- **version**: The S3 version of your firmware in your S3 bucket. This is different from your firmware version. You can find it by browsing to the Amazon S3 console, choosing your bucket, and on the top of the page, next to **Versions**, choosing **Show**.

**destination**

The destination for the signed firmware in an S3 bucket. The format of this parameter is the same as the source parameter.

**signing-material**

The ARN of your code-signing certificate. This ARN is generated when you import your certificate into ACM.

**signing-parameters**

A map of key-value pairs for signing. These can include any information that you want to use during signing.

**platform**

The platformId of the hardware platform to which you are distributing the OTA update.

To return a list of the available platforms and their platformId values, use the `aws signer list-signing-platforms` command.
The signing job starts and writes the signed firmware image into the destination Amazon S3 bucket. The file name for the signed firmware image is a GUID. You need this file name when you create a stream. You can find the generated file name by browsing to the Amazon S3 console and choosing your bucket. If you don’t see a file with a GUID file name, refresh your browser.

The command displays a job ARN and a job ID. You need these values later on. For more information about Code Signing for Amazon FreeRTOS, see Code Signing for Amazon FreeRTOS.

Signing Your Firmware Image Manually

Digitally sign your firmware image and upload your signed firmware image into your Amazon S3 bucket.

Creating a Stream of Your Firmware Update

The OTA Update service sends updates over MQTT messages. To do this, you must create a stream that contains your signed firmware update. Create a JSON file (stream.json) that identifies your signed firmware image. The JSON file should contain the following:

```
[
  {
    "fileId": "<your_file_id>",
    "s3Location": {
      "bucket": "<your_bucket_name>",
      "key": "<your_s3_object_key>"
    }
  }
]
```

The following list describes the attributes in the JSON file.

- **fileId**
  An arbitrary integer between 0 - 255 that identifies your firmware image.

- **s3Location**
  The bucket and key for the firmware to stream.

  - **bucket**
    The Amazon S3 bucket where your unsigned firmware image is stored.

  - **key**
    The file name of your signed firmware image in the Amazon S3 bucket. You can find this value in the Amazon S3 console by looking at the contents of your bucket. If you are using Code Signing for Amazon FreeRTOS, the file name is a GUID generated by Code Signing for Amazon FreeRTOS.

Use the `create-stream` CLI command to create a stream:

```
aws iot create-stream --stream-id <your_stream_id> --description <your_description> --files file://<stream.json> --role-arn <your_role_arn>
```

The following list describes the arguments for the `create-stream` CLI command.

- **stream-id**
  An arbitrary string to identify the stream.

- **description**
  An optional description of the stream.
files

One or more references to JSON files that contain data about firmware images to stream. The JSON file must contain the following attributes:

fileId

An arbitrary file ID.

s3Location

The bucket name where the signed firmware image is stored and the key (file name) of the signed firmware image.

bucket

The Amazon S3 bucket where the signed firmware image is stored.

key

The key (file name) of the signed firmware image. When you use Code Signing for Amazon FreeRTOS, this key is a GUID.

The following is an example stream.json file:

```
[
  {
    "fileId":123,
    "s3Location":{
      "bucket":"codesign-ota-bucket",
      "key":"48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
    }
  }
]
```

role-arn

An IAM role that grants access to the Amazon S3 bucket

To find the Amazon S3 object key of your signed firmware image, use the `aws signer describe-signing-job --job-id <my-job-id>` command where `my-job-id` is the job ID displayed by the `create-signing-job` CLI command. The output of the `describe-signing-job` command contains the key of the signed firmware image.

```
... text deleted for brevity ...
"signedObject": {
  "s3": {
    "bucketName": "ota-bucket",
    "key": "7309da2c-9111-48ac-8ee4-5a4262af4429"
  }
}
... text deleted for brevity ...
```

Creating an OTA Update

Use the `create-ota-update` CLI command to create an OTA update job:

```
aws iot create-ota-update --ota-update-id "<my OTA update>" --target-selection SNAPSHOT 
```
Note
Do not use any personally identifiable information (PII) in your OTA update job ID. Examples of personally identifiable information include:

- Your name
- Your IP address
- Your email address
- Your location
- Bank details
- Medical information

ota-update-id
An arbitrary OTA update ID.

target-selection
Valid values are:
- SNAPSHOT: The job terminates after deploying the update to the selected IoT thing or groups.
- CONTINUOUS: The job continues to deploy updates to devices added to the selected groups.

description
A text description of the OTA update.

files
One or more references to JSON files that contain data about the OTA update. The JSON file can contain the following attributes:
- fileName: The fully-qualified firmware image file name. For Texas Instruments CC3200SF-
LAUNCHXL, this must be "/sys/mcuflashimg.bin". For Microchip, this must be "mplab.production.bin"
- fileLocation: Contains information about the firmware update.
  - stream: The stream that contains the firmware update.
    - streamId: The stream ID specified in the create-stream CLI command.
    - fileId: The file ID specified in the JSON file passed to create-stream.
    - s3Location: The location in Amazon S3 of the firmware update.
    - bucket: The Amazon S3 bucket that contains the firmware update.
    - key: The firmware update key.
    - version: The firmware update version.
- codeSigning: Contains information about the code-signing job.
  - awsSignerJobId: The signer job ID generated by the start-signing-job command.
  - startSigningJobParameter: The information required to start a code-signing job.
    - signingProfileParameter: The information required for creating a signing job profile.
      - certificateArn: The ACM ARN of the certificate used to create a code-signing job.
      - platformId: The ID of the hardware platform you are using.
      - certificatePathOnDevice: The path to the certificate on your device.
    - signingProfileName: The signing profile name. If a profile with this name does not exist, you must provide values for signingProfileParameter. If a profile with the specified name exists, and you provide values for signingProfileParameter, the values you provide must match exactly the values you used for the signing profile.
  - destination: The location where the signed artifact is placed.
OTA Tutorial

- **s3Destination**: The Amazon S3 bucket where the signed artifact is placed.
- **bucket**: The Amazon S3 bucket.
- **prefix**: The prefix of the code-signing artifact. By default, this is `signedImage/`. This creates a folder called `signedImage` under your folder.

- **customCodeSigning**: Contains information about a custom signature.
  - **signature**: Contains a custom signature.
    - **inlineDocument**: The custom signature.
  - **certificateChain**: Contains a certificate chain for a custom signature.
    - **certificateName**: The path name of the code-signing certificate on the device.
    - **inlineDocument**: The certificate chain.
  - **hashAlgorithm**: The hash algorithm used to create the signature.
  - **signatureAlgorithm**: The signature algorithm used for code signing.
  - **attributes**: Arbitrary key/value pairs.

**targets**

One or more IoT thing ARNs that specify the devices to be updated by the OTA update.

**role-arn**

The ARN of your service role.

The following is an example of a JSON file passed into the `create-ota-update` command that uses Code Signing for Amazon FreeRTOS:

```json
[
  {
    "fileName": "firmware.bin",
    "fileLocation": {
      "stream": {
        "streamId": "004",
        "fileId":123
      }
    },
    "codeSigning": {
      "awsSignerJobId": "48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
    }
  }
]
```

The following is an example of a JSON file passed into the `create-ota-update` CLI command that uses an inline file to provide custom code-signing material:

```json
[
  {
    "fileName": "firmware.bin",
    "fileLocation": {
      "stream": {
        "streamId": "004",
        "fileId": 123
      }
    },
    "codeSigning": {
      "customCodeSigning":{
        "signature":{
          "inlineDocument":"<your_signature>"
        },
        "certificateChain":{
```
The following is an example of a JSON file passed into the `create-ota-update` CLI command that allows Amazon FreeRTOS OTA to start a code-signing job and create a code-signing profile and stream:

```
[
  {
    "fileName": "<your_firmware_path_on_device>",
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "<your_bucket_name>",
        "key": "<your_object_key>",
        "version": "<your_S3_object_version>"
      }
    },
    "codeSigning": {
      "startSigningJobParameter": {
        "signingProfileName": "myTestProfile",
        "signingProfileParameter": {
          "certificateArn": "<your_certificate_arn>",
          "platformId": "<your_platform_id>",
          "certificatePathOnDevice": "<certificate_path>"
        },
        "destination": {
          "s3Destination": {
            "bucket": "<your_destination_bucket>"
          }
        }
      }
    }
  }
]
```

The following is an example of a JSON file passed into the `create-ota-update` CLI command that creates an OTA update that starts a code signing job with an existing profile and uses the specified stream:

```
[
  {
    "fileName": "<your_firmware_path_on_device>",
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "<your_bucket_name>",
        "key": "<your_object_key>",
        "version": "<your_S3_object_version>"
      }
    },
    "codeSigning": {
      "startSigningJobParameter": {
        "signingProfileName": "<your_unique_profile_name>",
        "destination": {
          "s3Destination": {
            "bucket": "<your_destination_bucket>"
          }
        }
      }
    }
  }
]
The following is an example of a JSON file passed into the `create-ota-update` CLI command that allows Amazon FreeRTOS OTA to create a stream with an existing code-signing job ID:

```json
[
  {
    "fileName": "<your_firmware_path_on_device>",
    "fileVersion": "1",
    "codeSigning": {
      "awsSignerJobId": "<your_signer_job_id>
    }
  }
]
```

The following is an example of a JSON file passed into the `create-ota-update` CLI command that creates an OTA update. The update creates a stream from the specified S3 object and uses custom code signing:

```json
[
  {
    "fileName": "<your_firmware_path_on_device>",
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "<your_bucket_name>",
        "key": "<your_object_key>",
        "version": "<your_S3_object_version>"
      }
    },
    "codeSigning": {
      "customCodeSigning": {
        "signature": {
          "inlineDocument": "<your_signature>",
        },
        "certificateChain": {
          "inlineDocument": "<your_certificate_chain>",
          "certificateName": "<your_certificate_path_on_device>"
        },
        "hashAlgorithm": "<your_hash_algorithm>",
        "signatureAlgorithm": "<your_sig_algorithm>"
      }
    }
  }
]
```

You can use the `get-ota-update` CLI command to get the status of an OTA update:

```
aws iot get-ota-update --ota-update-id <your-ota-update-id>
```

This command returns one of the following values:

- **CREATE_PENDING**
  
  The creation of an OTA update is pending.

- **CREATE_IN_PROGRESS**
  
  An OTA update is being created.
CREATE_COMPLETE

An OTA update has been created.

CREATE_FAILED

The creation of an OTA update failed.

DELETE_IN_PROGRESS

An OTA update is being deleted.

DELETE_FAILED

The deletion of an OTA update failed.

Listing OTA Updates

You can use the list-ota-updates CLI command to get a list of all OTA updates by:

```
aws iot list-ota-updates
```

The output from the list-ota-updates command looks like this:

```json
{
  "otaUpdates": [
    {
      "otaUpdateId": "my_ota_update2",
      "creationDate": 1522778769.042
    },
    {
      "otaUpdateId": "my_ota_update1",
      "creationDate": 1522775938.956
    },
    {
      "otaUpdateId": "my_ota_update",
      "creationDate": 1522775151.031
    }
  ]
}
```

Getting Information About an OTA Update

You can use the get-ota-update CLI command to get information about a specific OTA update:

```
aws iot get-ota-update --ota-update-id <my-ota-update-id>
```

The output from the get-ota-update command looks like this:

```json
{
  "otaUpdateInfo": {
    "otaUpdateId": "myotaupdate1",
    "creationDate": 1522444438.424,
    "lastModifiedDate": 1522444440.681,
  }
}
```
Deleting OTA-Related Data

Currently, you cannot use the AWS IoT console to delete streams or OTA updates. You can use the AWS CLI to delete streams, OTA updates, and the IoT jobs created during an OTA update.

Deleting an OTA Stream

When you create an OTA update either you or the AWS IoT console creates a stream to break the firmware up into chunks so it can be sent over MQTT. You can delete this stream with the `delete-stream` CLI command. For example:

```
aws iot delete-stream --stream-id <your_stream_id>
```

Deleting an OTA Update

When you create an OTA update, these things are created:

- An entry in the OTA update job database.
- An AWS IoT job to perform the update.
- An AWS IoT job execution for each device being updated.

The `delete-ota-update` command deletes the entry in the OTA update job database only. You must use the `delete-job` command to delete the AWS IoT job.

Use the `delete-ota-update` command to delete an OTA update:

```
aws iot delete-ota-update --ota-update-id <your_ota_update_id>
```

ota-update-id

The ID of the OTA update to delete.
OTA Update Manager Service

The OTA Update Manager service provides a way to:

- Create an OTA update.
- Get information about an OTA update.
- List all OTA updates associated with your AWS account.

**delete-stream**

Deletes the stream associated with the OTA update.

**force-delete-aws-job**

Deletes the AWS IoT job associated with the OTA update. If this flag is not set and the job is in the In_Progress state, the job is not deleted.

Deleting an IoT Job Created for an OTA Update

Amazon FreeRTOS creates an AWS IoT job when you create an OTA update. A job execution is also created for each device that processes the job. You can use the `delete-job` CLI command to delete a job and its associated job executions:

```bash
aws iot delete-job --job-id <your-job-id> --no-force
```

The `--no-force` parameter specifies that only jobs that are in a terminal state (COMPLETED or CANCELLED) can be deleted. You can delete a job that is in a non-terminal state by passing the `--force` parameter. For more information, see DeleteJob API.

**Note**

Deleting a job with a status of IN_PROGRESS interrupts any job executions that are IN_PROGRESS on your devices and can result in a device being left in a nondeterministic state. Make sure that each device executing a job that has been deleted can recover to a known state.

Depending on the number of job executions created for the job and other factors, a few minutes to delete a job. While the job is being deleted, the status of the job appears as DELETION_IN_PROGRESS. Attempting to delete or cancel a job whose status is already DELETION_IN_PROGRESS results in an error.

You can use the `delete-job-execution` to delete a job execution. You might want to delete a job execution when a small number of devices are unable to process a job. This deletes the job execution for a single device. For example:

```bash
aws iot delete-job-execution --job-id <your-job-id> --thing-name <your-thing-name> --execution-number <your-job-execution-number> --no-force
```

As with the `delete-job` CLI command, you can pass the `--force` parameter to the `delete-job-execution` to force the deletion of an execution job execution. For more information, see DeleteJobExecution API.

**Note**

Deleting a job execution with a status of IN_PROGRESS interrupts any job executions that are IN_PROGRESS on your devices and can result in a device being left in a nondeterministic state. Make sure that each device executing a job that has been deleted is able to recover to a known state.

For more information about using the OTA update demo application, see OTA Demo Application (p. 168).
• Delete an OTA update.

An OTA update is a data structure maintained by the OTA Update Manager service. It contains:

• An OTA update ID.
• An optional OTA update description.
• A list of devices to update (targets).
• The type of OTA update: CONTINUOUS or SNAPSHOT.
• A list of files to send to the target devices.
• An IAM role that allows access to the AWS IoT Jobs service.
• An optional list of user-defined name-value pairs.

OTA updates were designed to be used to update device firmware, but you can use them to send any files you want to one or more devices registered with AWS IoT. When you send files over the air, it is best practice to digitally sign them so the devices that receive the files can verify they have not been tampered with en route. You can sign your files with Code Signing for Amazon FreeRTOS or you can use your own code-signing tools.

After your files have been digitally signed, you use the Amazon Streaming service to create a stream. The service breaks up your files into blocks that can be sent over MQTT to your devices.

When you create an OTA update, the OTA Manager service creates an AWS IoT job to notify your devices an update is available. The Amazon FreeRTOS OTA agent runs on your devices and listens for update messages. When an update is available, it streams the update over MQTT and stores the files locally. It checks the digital signature of the downloaded files and if valid, installs the firmware update. If you are not using Amazon FreeRTOS, you must implement your own OTA agent to listen for and download updates and perform any installation operations.

Integrating the OTA Agent into Your Application

The OTA agent is designed to simplify the amount of code you must write to add OTA update functionality to your product. That integration burden consists primarily of initialization of the OTA agent and, optionally, creating a custom callback function for responding to the OTA completion event messages.

**Note**
Although the integration of the OTA update feature into your application is rather simple, the OTA update system requires an understanding of more than just device code integration. To familiarize yourself with how to configure your AWS account with AWS IoT things, credentials, code-signing certificates, provisioning devices, and OTA update jobs, see Amazon FreeRTOS Prerequisites.

MQTT Connection Management

The OTA agent uses the MQTT protocol for all of its communication with AWS IoT services, but it does not manage the MQTT connection. To assure that the OTA agent does not interfere with the connection management policy of your application, the MQTT connection, including disconnect and any reconnect functionality, must be handled by the main "user" application.

Simple OTA Demo

The following is an excerpt of a simple OTA demo that shows how the agent connects to the MQTT broker and initializes the OTA agent. In this example, we configure the demo to use the default OTA
completion callback and simply print out some statistics once per second. For brevity, we leave out some
details from this demo.

For a working example that uses the AWS IoT MQTT broker, see the OTA demo code.

Because the OTA agent is its own task, the intentional one-second delay in this example affects this
application only. It has no impact on the performance of the agent.

```c
/* Create the MQTT Client. */
if( MQTT_AGENT_Create( &xMQTT_h ) == eMQTTAgentSuccess )
{
    for ( ; ; )
    {
        memset( &xConnParm, 0, sizeof( xConnParm );
        /* ... Set MQTT connection parameters here per your application needs ... */
        configPRINTF( ( "Connecting to %s\r\n", clientcredentialMQTT_BROKER_ENDPOINT ) );
        if( MQTT_AGENT_Connect( xMQTT_h, &xConnParm, myappMAX_AWS_CONNECT_WAIT_IN_TICKS ) == eMQTTAgentSuccess )
        {
            configPRINTF( ( "Connected to broker.\r\n" ) );
            /* Initialize the OTA Agent with the default completion callback handler. */
            OTA_AgentInit( xMQTT_h, ( const uint8_t * ) ( clientcredentialIOT_THING_NAME ), NULL,
                /* NULL uses the default
callback handler. */ ( TickType_t ) ~0 );
            while( ( eState = OTA_GetAgentState() ) != eOTA_AgentState_NotReady )
            {
                /* Wait forever for OTA traffic but allow other tasks to run
and output statistics only once per second. */
                vTaskDelay( myappONE_SECOND_DELAY_IN_TICKS );
                configPRINTF( ( "State: %s  Received: %u   Queued: %u   Processed: %u
Dropped: %u\r\n", pcStateStr[eState],
                    OTA_GetPacketsReceived(),
                    OTA_GetPacketsQueued(),
                    OTA_GetPacketsProcessed(),
                    OTA_GetPacketsDropped() ) );
            }
            /* ... Handle MQTT disconnect per your application needs ... */
        }
        else
        {
            configPRINTF( ( "ERROR: MQTT_AGENT_Connect() Failed.\r\n" ) );
            /* After failure to connect or a disconnect, wait an arbitrary one second before
retry. */
            vTaskDelay( myappONE_SECOND_DELAY_IN_TICKS );
        }
    }
    else
    {
        configPRINTF( ( "Failed to create MQTT client.\r\n" ) );
    }
}
```

Here is the high-level flow of this demo application:

- Create an MQTT agent context.
- Connect to your AWS IoT endpoint.
- Initialize the OTA agent.
- Loop allowing an OTA update job and output statistics once a second.
- If the agent stops, wait one second and try connecting again.

## Using a Custom Callback for OTA Completion Events

The previous example used the built-in callback handler for OTA completion events by specifying NULL for the third parameter to the `OTA_AgentInit` API. If you want to implement custom handling of the completion events, you must pass the function address of your callback handler to the `OTA_AgentInit` API. During the OTA process, the agent can send one of the following event enums to the callback handler. It is up to the application developer to decide how and when to handle these events.

```c
/**
 * @brief OTA Job callback events.
 *
 * After an OTA update image is received and authenticated, the agent calls the user
 * callback (set with the OTA_AgentInit API) with the value eOTA_JobEvent_Activate to
 * signal that the device must be rebooted to activate the new image. When the device
 * boots, if the OTA job status is in self test mode, the agent calls the user callback
 * with the value eOTA_JobEvent_StartTest, signaling that any additional self tests
 * should be performed.
 *
 * If the OTA receive fails for any reason, the agent calls the user callback with
 * the value eOTA_JobEvent_Fail instead to allow the user to log the failure and take
 * any action deemed appropriate by the user code.
 *
 */
typedef enum {
   eOTA_JobEvent_Activate,  /*! OTA receive is authenticated and ready to activate. */
   eOTA_JobEvent_Fail,      /*! OTA receive failed. Unable to use this update. */
   eOTA_JobEvent_StartTest  /*! OTA job is now in self test, perform user tests. */
} OTA_JobEvent_t;
```

The OTA agent can receive an update in the background during active processing of the main application. The purpose of delivering these events is to allow the application to decide if action can be taken immediately or if it should be deferred until after completion of some other application-specific processing. This prevents an unanticipated interruption of your device during active processing (for example, vacuuming) that would be caused by a reset after a firmware update. These are the job events received by the callback handler:

- **eOTA_JobEvent_Activate event**
  
  When this event is received by the callback handler, you can either reset the device immediately or schedule a call to reset the device later. This allows you to postpone the device reset and self-test, if necessary.

- **eOTA_JobEvent_Fail event**
  
  When this event is received by the callback handler, the update has failed. You do not need to do anything in this case. You might want to output a log message or do something application-specific.

- **eOTA_JobEvent_StartTest event**
  
  The self test phase is meant to allow newly updated firmware to execute and test itself before determining that it is properly functioning and commit it to be the latest permanent application image. When a new update is received and authenticated and the device has been reset, the OTA agent will send the `eOTA_JobEvent_StartTest` event to the callback function when it is ready for testing. The developer may choose to add any tests deemed required to determine if the device firmware is functioning properly after update. When the device firmware is deemed reliable by the self tests, the code must commit the firmware as the new permanent image by calling the `OTA_SetImageState( eOTA_ImageState_Accepted )` function.
If your device has no special hardware or mechanisms that need to be tested, you can use the default callback handler. Upon receipt of the eOTA_JobEvent_Activate event, the default handler resets the device immediately.

**OTA Security**

The following are three aspects of OTA security:

**Connection security**

The OTA Update Manager relies on existing security mechanisms, like TLS mutual authentication, used by AWS IoT. OTA update traffic passes through the AWS IoT device gateway and uses AWS IoT security mechanisms. Each incoming and outgoing MQTT message through the device gateway undergoes strict authentication and authorization.

**Authenticity and integrity of OTA updates**

Firmware can be digitally signed before an OTA update to ensure that it is from a reliable source and has not been tampered with. The Amazon FreeRTOS OTA Update Manager uses the Code Signing for Amazon FreeRTOS to automatically sign your firmware. For more information, see Code Signing for Amazon FreeRTOS. The OTA agent, which runs on your devices, performs integrity checks on the firmware when it arrives on the device.

**Operator security**

Every API call made through the control plane API undergoes standard IAM Signature Version 4 authentication and authorization. To create a deployment, you must have permissions to invoke the CreateDeployment, CreateJob, and CreateStream APIs. In addition, in your Amazon S3 bucket policy or ACL, you must give read permissions to the AWS IoT service principal so that the firmware update stored in Amazon S3 can be accessed during streaming.

**Code Signing for Amazon FreeRTOS**

The AWS IoT console uses Code Signing for Amazon FreeRTOS to automatically sign your firmware image for any device supported by AWS IoT.

Code Signing for Amazon FreeRTOS uses a certificate and private key that you import into ACM. You can use a self–signed certificate for testing, but we recommend that you obtain a certificate from a well–known commercial certificate authority (CA).

Code–signing certificates use the X.509 version 3 Key Usage and Extended Key Usage extensions. The Key Usage extension is set to Digital Signature and the Extended Key Usage extension is set to Code Signing. For more information about signing your code image, see the Code Signing for Amazon FreeRTOS Developer Guide and the Code Signing for Amazon FreeRTOS API Reference.

**Note**

You can download the Code Signing for Amazon FreeRTOS SDK from https://tools.signer.aws.a2z.com/awssigner-tools-v2.zip.

**OTA Troubleshooting**

The following sections contain information to help you troubleshoot issues with OTA updates.

**Topics**

- Setting Up Cloudwatch Logs for OTA Updates (p. 148)
- Logging AWS IoT OTA API Calls with AWS CloudTrail (p. 151)
- Troubleshooting OTA Updates with the Texas Instruments CC3220SF Launchpad (p. 153)
Setting Up Cloudwatch Logs for OTA Updates

The OTA Update service supports logging with Amazon CloudWatch. You can use the AWS IoT console to enable and configure Amazon CloudWatch logging for OTA updates. For more information about CloudWatch Logs, see Cloudwatch Logs.

To enable logging, you must create an IAM role and configure OTA update logging.

Note
Before you enable OTA update logging, make sure you understand the CloudWatch Logs access permissions. Users with access to CloudWatch Logs can see your debugging information. For information, see Authentication and Access Control for Amazon CloudWatch Logs.

Create a Logging Role and Enable Logging

Use the AWS IoT console to create a logging role and enable logging.

1. From the navigation pane, choose Settings.
2. Under Logs, choose Edit.
3. Under Level of verbosity, choose Debug.
4. Under Set role, choose Create new to create an IAM role for logging.
5. Under Name, enter a unique name for your role. Your role will be created with all required permissions.
6. Choose Update.

OTA Update Logs

The OTA Update service publishes logs to your account when one of the following occurs:

- An OTA update is created.
- An OTA update is completed.
- A code-signing job is created.
- A code-signing job is completed.
- An AWS IoT job is created.
- An AWS IoT job is completed.
- A stream is created.

You can view your logs in the CloudWatch console.

To view an OTA Update in CloudWatch Logs

1. From the navigation pane, choose Logs.
2. In Log Groups, choose AWSIoTLogsV2.

OTA update logs can contain the following properties:

accountld

- The AWS account ID in which the log was generated.

actionType

- The action that generated the log. This can be set to one of the following values:
• CreateOTAUpdate: An OTA update was created.
• DeleteOTAUpdate: An OTA update was deleted.
• StartCodeSigning: A code-signing job was started.
• CreateAWSJob: An AWS IoT job was created.
• CreateStream: A stream was created.
• GetStream: A request for a stream was sent to the AWS IoT Streaming service.
• DescribeStream: A request for information about a stream was sent to the AWS IoT Streaming service.

awsJobId

The AWS IoT job ID that generated the log.

clientId

The MQTT client ID that made the request that generated the log.

clientToken

The client token associated with the request that generated the log.

details

Additional information about the operation that generated the log.

logLevel

The logging level of the log. For OTA update logs, this is always set to DEBUG.

otaUpdateId

The ID of the OTA update that generated the log.

protocol

The protocol used to make the request that generated the log.

status

The status of the operation that generated the log. Valid values are:
• Success
• Failure

streamId

The AWS IoT stream ID that generated the log.

timestamp

The time when the log was generated.

topicName

An MQTT topic used to make the request that generated the log.

**Example Logs**

The following is an example log generated when a code-signing job is started:

```
{
  "timestamp": "2018-07-23 22:59:44.955",
  "logLevel": "DEBUG",
  "awsJobId": "example-job-id",
  "clientToken": "example-token",
  "otaUpdateId": "example-update-id",
  "streamId": "example-stream-id",
  "topicName": "example-topic",
  "details": "Example details"
}
```
The following is an example log generated when an AWS IoT job is created:

{  
  "timestamp": "2018-07-23 22:59:45.363",
  "logLevel": "DEBUG",
  "accountId": "123456789012",
  "status": "Success",
  "actionType": "CreateAWSJob",
  "otaUpdateId": "08957b03-eea3-448a-87fe-743e6891ca3a",
  "awsJobId": "08957b03-eea3-448a-87fe-743e6891ca3a",
  "details": "Create AWS Job. The request status is SUCCESS."
}

The following is an example log generated when an OTA update is created:

{  
  "timestamp": "2018-07-23 22:59:45.413",
  "logLevel": "DEBUG",
  "accountId": "123456789012",
  "status": "Success",
  "actionType": "CreateOTAUpdate",
  "otaUpdateId": "08957b03-eea3-448a-87fe-743e6891ca3a",
  "details": "OTAUpdate creation complete. The request status is SUCCESS."
}

The following is an example log generated when a stream is created:

{  
  "timestamp": "2018-07-23 23:00:26.391",
  "logLevel": "DEBUG",
  "accountId": "123456789012",
  "status": "Success",
  "actionType": "CreateStream",
  "otaUpdateId": "3d3dc5f7-3d6d-47ac-9252-45821ac7cfb0",
  "streamId": "6be2303d-3637-48f0-ace9-0b87b1b9a824",
  "details": "Create stream. The request status is SUCCESS."
}

The following is an example log generated when an OTA update is deleted:

{  
  "timestamp": "2018-07-23 23:03:09.505",
  "logLevel": "DEBUG",
  "accountId": "123456789012",
  "status": "Success",
  "actionType": "DeleteOTAUpdate",
  "otaUpdateId": "9bdd78fb-f113-4001-9675-1b595982292f",
  "details": "Delete OTA Update. The request status is SUCCESS."
}
The following is an example log generated when a device requests a stream from the streaming service:

```
{
  "timestamp": "2018-07-25 22:09:02.678",
  "logLevel": "DEBUG",
  "accountId": "123456789012",
  "status": "Success",
  "actionType": "GetStream",
  "protocol": "MQTT",
  "clientId": "b9d2e49c-94fe-4ed1-9b07-286afed7e4c8",
  "topicName": "/aws/things/b9d2e49c-94fe-4ed1-9b07-286afed7e4c8/stream/1e51e9a8-9a4c-4c50-b005-d38452a956af/get/json",
  "streamId": "1e51e9a8-9a4c-4c50-b005-d38452a956af",
  "details": "The request status is SUCCESS."
}
```

The following is an example log generated when a device calls the DescribeStream API:

```
{
  "timestamp": "2018-07-25 22:10:12.690",
  "logLevel": "DEBUG",
  "accountId": "123456789012",
  "status": "Success",
  "actionType": "DescribeStream",
  "protocol": "MQTT",
  "clientId": "581075e0-4639-48ee-8b94-2cf304168e43",
  "topicName": "/aws/things/581075e0-4639-48ee-8b94-2cf304168e43/stream/71c101a8-bcc5-4929-9fe2-af563af0c139/describe/json",
  "streamId": "71c101a8-bcc5-4929-9fe2-af563af0c139",
  "clientToken": "clientToken",
  "details": "The request status is SUCCESS."
}
```

Logging AWS IoT OTA API Calls with AWS CloudTrail

Amazon FreeRTOS is integrated with CloudTrail, a service that captures all of the AWS IoT OTA API calls and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from your code to the AWS IoT OTA APIs. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT OTA, the source IP address from which the request was made, who made the request, when it was made, and so on.

To learn more about CloudTrail, including how to configure and enable it, see the [AWS CloudTrail User Guide](#).

Amazon FreeRTOS Information in CloudTrail

When CloudTrail logging is enabled in your AWS account, most API calls made to AWS IoT OTA actions are tracked in CloudTrail log files where they are written with other AWS service records. CloudTrail determines when to create and write to a new file based on a time period and file size.

**Note**

AWS IoT OTA data plane actions (device side) are not logged by CloudTrail. Use CloudWatch to monitor these.

AWS IoT OTA control plane actions are logged by CloudTrail. For example, calls to the CreateOTAUpdate, GetOTAUpdate, and CreateStream sections generate entries in the CloudTrail log files.
Every log entry contains information about who generated the request. The user identity information in the log entry helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element. AWS OTA IoT actions are documented in the AWS IoT OTA API Reference.

You can store your log files in your Amazon S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

If you want to be notified upon log file delivery, you can configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate AWS IoT OTA log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket.

For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

Understanding Amazon FreeRTOS Log File Entries

CloudTrail log files can contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. Log entries are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the log from a call to CreateOTAUpdate action.

```json
{
   "eventVersion": "1.05",
   "userIdentity": {
      "type": "IAMUser",
      "principalId": "EXAMPLE",
      "arn": "arn:aws:iam::<your_aws_account>::user/<your_user_id>",
      "accountId": "<your_aws_account>",
      "accessKeyId": "<your_access_key_id>",
      "userName": "<your_username>",
      "sessionContext": {
         "attributes": {
            "mfaAuthenticated": "false",
            "creationDate": "2018-08-23T17:27:08Z"
         }
      },
      "invokedBy": "apigateway.amazonaws.com"
   },
   "eventTime": "2018-08-23T17:27:19Z",
   "eventSource": "iot.amazonaws.com",
   "eventName": "CreateOTAUpdate",
   "awsRegion": "<your_aws_region>",
   "sourceIPAddress": "apigateway.amazonaws.com",
   " userAgent": "apigateway.amazonaws.com",
   "requestParameters": {
      "targets": [
         "arn:aws:iot::<your_aws_region>::<your_aws_account>::thing/Thing_CMH"
      ]
   }
}
Troubleshooting OTA Updates with the Texas Instruments CC3220SF Launchpad

The CC3220SF Launchpad platform provides a software tamper detection mechanism that uses a security-alert counter that is incremented whenever there is an integrity violation. The device is locked when the security-alert counter reaches a pre-determined threshold (the default is 15) and the host receives the asynchronous event SL_ERROR_DEVICE_LOCKED_SECURITY_ALERT. The locked device will then have limited accessibility. To recover the device, you can reprogram it or use the “restore-to-factory” process to revert to the factory image. You should program the desired behavior by updating the asynchronous event handler in “network_if.c”. For more information, see Texas Instruments SimpleLink CC3120, CC3220 Wi-Fi Internet-on-a-chip Solution Built-In Security Features Application Report.

Amazon FreeRTOS Console User Guide

Managing Amazon FreeRTOS Configurations

You can use the Amazon FreeRTOS console to manage software configurations and download Amazon FreeRTOS software for your device. The Amazon FreeRTOS software is prequalified on a variety of platforms. It includes the required hardware drivers, libraries, and example projects to help get you started quickly. You can choose between predefined configurations or create custom configurations.

Predefined Amazon FreeRTOS Configurations

Predefined configurations are defined for the prequalified platforms:

- TI CC3220SF-LAUNCHXL
- STM32 IoT Discovery Kit
• NXP LPC54018 IoT Module
• Microchip Curiosity PIC32MZEF
• Espressif ESP32-DevKitC
• Espressif ESP32-WROVER-KIT
• Infineon XMC4800 IoT Connectivity Kit
• Xilinx Avnet MicroZed Industrial IoT Starter Kit
• FreeRTOS Windows Simulator

The predefined configurations make it possible for you to get started quickly with the supported use cases without thinking about which libraries are required. To use a predefined configuration, browse to the Amazon FreeRTOS console, find the configuration you want to use, and then choose Download.

You can also customize a predefined configuration if you want to change the Amazon FreeRTOS version, hardware platform, or libraries of the configuration. Customizing a predefined configuration creates a new custom configuration and does not overwrite the predefined configuration in the Amazon FreeRTOS console.

To create a custom configuration from a predefined configuration

1. Browse to the Amazon FreeRTOS console.
2. In the navigation pane, choose Software.
4. Choose the ellipsis next to the predefined configuration that you want to customize, and then choose Customize.
5. On the Configure Amazon FreeRTOS Software page, choose the Amazon FreeRTOS version, hardware platform, and libraries, and give the new configuration a name and a description.
6. At the bottom of the page, choose Create and download to create and download the custom configuration.

Custom Amazon FreeRTOS Configurations

Custom configurations allow you to specify your hardware platform, integrated development platform (IDE), compiler, and only those RTOS libraries you require. This leaves more space on your devices for application code.

To create a custom configuration

1. Browse to the Amazon FreeRTOS console and choose Create new.
2. Select the version of Amazon FreeRTOS that you want to use. The latest version is used by default.
3. On the New Software Configuration page, choose Select a hardware platform, and choose one of the prequalified platforms.
4. Choose the IDE and compiler you want use.
5. For the Amazon FreeRTOS libraries you require, choose Add Library. If you choose a library that requires another library, it is added for you. If you want to choose more libraries, choose Add another library.
6. In the Demo Projects section, enable one of the demo projects. This enables the demo in the project files.
7. In Name required, enter a name for your custom configuration.
   
   **Note**
   Do not use any personally identifiable information in your custom configuration name.
8. In Description, enter a description for your custom configuration.
9. At the bottom of the page, choose Create and download to create and download your custom configuration.

To edit a custom configuration

1. Browse to the Amazon FreeRTOS console.
2. In the navigation pane, choose Software.
4. Choose the ellipsis next to the configuration you want to edit, and then choose Edit.
5. On the Configure Amazon FreeRTOS Software page, you can change your configuration's Amazon FreeRTOS version, hardware platform, libraries, and description.
6. At the bottom of the page, choose Save and download to save and download the configuration.

To delete a custom configuration

1. Browse to the Amazon FreeRTOS console.
2. In the navigation pane, choose Software.
4. Choose the ellipsis next to the configuration you want to delete, and then choose Delete.
Amazon FreeRTOS Demo Projects

This section contains resources that are useful after you have a basic understanding of Amazon FreeRTOS. If you haven’t already, we recommend that you first read the Getting Started with Amazon FreeRTOS (p. 4).

Topics

• Navigating the Demo Applications (p. 156)
• Bluetooth Low Energy Demo Applications (Beta) (p. 157)
• Secure Sockets Echo Client Demo (p. 165)
• Device Shadow Demo Application (p. 166)
• Greengrass Discovery Demo Application (p. 167)
• OTA Demo Application (p. 168)
• Demo Bootloader for the Microchip Curiosity PIC32MZEF (p. 171)

Navigating the Demo Applications

This section contains information about directory and file organization and configuration files for the demos.

Directory and File Organization

There are two subfolders in the main Amazon FreeRTOS directory:

• demos

Contains example code that can be run on an Amazon FreeRTOS device to demonstrate Amazon FreeRTOS functionality. There is one subdirectory for each target platform selected. These subdirectories contain code used by the demos, but not all demos can be run independently. If you use the Amazon FreeRTOS console, only the target platform you choose has a subdirectory under demos.

The function DEMO_RUNNER_RunDemos() located in AmazonFreeRTOS\demos\common \demo_runner\aws_demo_runner.c contains code that calls each example. By default, only the vStartMQTTEchoDemo() function is called. Depending on the configuration you selected when you downloaded the code, or whether you obtained the code from GitHub, the other example runner functions are either commented out or omitted entirely. Although you can change the selection of demos here, be aware that not all combinations of examples work together. Depending on the combination, the software might not be able to be executed on the selected target due to memory constraints. All of the examples that can be executed by Amazon FreeRTOS appear in the common directory under demos.

• lib

The lib directory contains the source code of the Amazon FreeRTOS libraries. These libraries are available to you as part of Amazon FreeRTOS:

• MQTT
• Device shadow
• Greengrass
• Wi-Fi
There are helper functions that assist in implementing the library functionality. We do not recommend that you change these helper functions. If you need to change one of these libraries, make sure it conforms to the library interface defined in the `libs/include` directory.

**Configuration Files**

The demos have been configured to get you started quickly. You might want to change some of the configurations for your project to create a version that runs on your platform. You can find configuration files at `AmazonFreeRTOS/<vendor>/platform/common/config_files`.

The configuration files include:

- `aws_bufferpool.h`
  - Configures the size and quantity of static buffers available for use by the application.
- `aws_clientcredential_keys.h`
  - Configures your device certificate and private key.
- `aws_demo_config.h`
  - Configures the task parameters used in the demos: stack size, priorities, and so on.
- `aws_ggd_config.h`
  - Configures the parameters used to configure a Greengrass core, such as network interface IDs.
- `aws_mqtt_agent_config.h`
  - Configures the parameters related to MQTT operations, such as task priorities, MQTT brokers, and keep-alive counts.
- `aws_mqtt_library.h`
  - Configures MQTT library parameters, such as the subscription length and the maximum number of subscriptions.
- `aws_secure_sockets_config.h`
  - Configures the timeouts and the byte ordering when using SSL.
- `aws_shadow_configure.h`
  - Configures the parameters used for an AWS IoT shadow, such as the number of JSMN tokens used when parsing a JSON file received from a shadow.
- `aws_clientcredential.h`
  - Configures parameters, including the Wi-Fi (SSID, password, and security type), the MQTT broker endpoint, and IoT thing name.
- `FreeRTOSConfig.h`
  - Configures the FreeRTOS kernel for multitasking operations.

**Bluetooth Low Energy Demo Applications (Beta)**

The Bluetooth Low Energy (BLE) Library is in public beta release for Amazon FreeRTOS and is subject to change.
Overview

Amazon FreeRTOS BLE includes three demo applications:

**MQTT over BLE (p. 160) Demo**
This application demonstrates how to use the MQTT over BLE service.

**Wi-Fi Provisioning (p. 162) Demo**
This application demonstrates how to use the Wi-Fi Provisioning service.

**Generic Attributes Server (p. 164) Demo**
This application demonstrates how to use the Amazon FreeRTOS BLE middleware APIs to create a simple GATT server.

Prerequisites

To follow along with these demos, you need a microcontroller with Bluetooth Low Energy capabilities.

Before you begin, do the following:

**Set Up AWS IoT**
To set up AWS IoT, you need to do the following:

- Set up an AWS account.
- Register your device as an AWS IoT thing.
- Download your AWS IoT credentials.

For more information about setting up AWS IoT, see the [AWS IoT Developer Guide](https://aws.amazon.com/iot/).

**Set Up Amazon Cognito**
To set up Amazon Cognito, you need to do the following:

- Set up an AWS account.
- Create an Amazon Cognito user pool.
- Create an Amazon Cognito identity pool.
- Attach an IAM policy to the authenticated identity.

For more information about setting up Amazon Cognito, see the [Amazon Cognito Developer Guide](https://aws.amazon.com/cognito/).

**Set Up Your Environment**
To set up your environment, do the following:

- Set up your microcontroller's environment with Amazon FreeRTOS and the Amazon FreeRTOS BLE library. You can download Amazon FreeRTOS from [GitHub](https://github.com/AmazonWebServices/amazon-freertos).
For information about getting started with Amazon FreeRTOS on an Amazon FreeRTOS-qualified microcontroller, see information for your board in Getting Started with Amazon FreeRTOS.

Note
You can run the demos on any BLE-enabled microcontroller with Amazon FreeRTOS and ported Amazon FreeRTOS BLE libraries. Currently, the Amazon FreeRTOS MQTT over BLE (p. 160) demo project is fully ported to the following BLE-enabled devices:
- STMicroelectronics STM32L4 Discovery Kit IoT Node, with the STBTLE-1S BLE module
- Espressif ESP32-DevKitC and the ESP-WROVER-KIT
- Nordic nRF52840-DK

- Install the Amazon FreeRTOS BLE Mobile SDK Demo Application (p. 159) on your Android or iOS device. The demo application is a common component of the demos.

For information about installing the demo app, see the GitHub README files for the Amazon FreeRTOS BLE Mobile SDK for Android or the Amazon FreeRTOS BLE Mobile SDK for iOS.

Common Components

The Amazon FreeRTOS demo applications have two common components:

- Network Manager
- BLE Mobile SDK demo application

Network Manager

Network Manager manages your microcontroller’s network connection. It is located in your Amazon FreeRTOS directory at \demos\common\network_manager\aws_iot_network_manager.c. If the network manager is enabled for both Wi-Fi and BLE, the demos start with BLE by default. If the BLE connection is disrupted, and your board is Wi-Fi-enabled, the Network Manager switches to an available Wi-Fi connection to prevent you from disconnecting from the network.

To enable a network connection type with the Network Manager, add the network connection type to the configENABLED_NETWORKS parameter in demos/vendor/board/common/config_files/aws_iot_network_config.h. For example, if you have both BLE and Wi-Fi enabled, the line that starts with #define configENABLED_NETWORKS in aws_iot_network_config.h reads as follows:

```c
#define configENABLED_NETWORKS ( AWSIOT_NETWORK_TYPE_BLE | AWSIOT_NETWORK_TYPE_WIFI )
```

To get a list of currently supported network connection types, see \lib\include aws_iot_network_manager.h.

Amazon FreeRTOS BLE Mobile SDK Demo Application

Each demo uses the Amazon FreeRTOS BLE Mobile SDK demo application, which can be found in the BLE Android SDK or the BLE iOS SDK under FreeRTOSDemo/Examples. In this example, we use the iOS version of the demo mobile application.

To discover and establish secure connections with your microcontroller across BLE with the demo mobile application, for each demo, do the following:

1. Run the MQTT over BLE (p. 160), Wi-Fi Provisioning (p. 162), or Generic Attributes Server (p. 164) demo on your microcontroller.
2. Start the BLE mobile SDK demo application on your mobile device.
3. Confirm that your microcontroller appears under **Devices** on the BLE mobile SDK demo app.

![Device List]

**Note**  
Only devices with Amazon FreeRTOS and the device information service (`\lib\bluetooth_low_energy\services\device_information`) appear in the list.

4. Choose your microcontroller from the list of devices. The application establishes a connection with the board, and a green line appears next to the connected device.

![Connected Device]

You can disconnect from your microcontroller by dragging the line to the left.

5. You might be prompted to pair your microcontroller and mobile device.

![Pairing Request]

If the code for numeric comparison is the same on both devices, pair the devices.

**Note**  
The BLE Mobile SDK demo application uses Amazon Cognito for user authentication. Make sure that you have set up a Amazon Cognito user and identity pools, and that you have attached IAM policies to authenticated identities.

### MQTT over BLE

In the MQTT over BLE demo, your microcontroller publishes messages to the AWS IoT cloud through an MQTT proxy.
To subscribe to a demo MQTT topic

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose Test to open the MQTT client.
3. In Subscription topic, enter freertos/demos/echo, and then choose Subscribe to topic.

You can run the MQTT demo across a BLE or Wi-Fi connection. The configuration of the Network Manager (p. 159) determines which connection type is used.

If you use BLE to pair the microcontroller with your mobile device, the MQTT messages are routed through the BLE mobile SDK demo application on your mobile device.

If you use Wi-Fi, the demo is the same as the MQTT Hello World demo project located in demos/vendor/board/ide. That demo is used in most of the Getting Started with Amazon FreeRTOS demo projects.

To enable the demo

If you have already enabled the demo by following the instructions in the getting started guide for your device, you can skip these instructions.

1. Confirm that the MQTT over BLE and Wi-Fi Provisioning services are enabled in lib/defs/aw_sbe_services_init.c. The services are enabled by default.
2. Open demos/common/demo_runner/aws_demo_runner.c, and in the demo declarations, uncomment extern void vStartMQTTBLEEchoDemo( void );. In the DEMO_RUNNER_RunDemos definition, uncomment vStartMQTTBLEEchoDemo();.

To run the demo

If the Network Manager is configured for Wi-Fi only, simply build and run the demo project on your board.

If the Network Manager is configured for BLE, do the following:

1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your board and your mobile device using the Amazon FreeRTOS BLE Mobile SDK Demo Application (p. 159).
3. From the Devices list in the demo mobile app, choose your microcontroller, and then choose MQTT Proxy to open the MQTT proxy settings.
4. Touch **Enable MQTT proxy** to enable the MQTT proxy. The slider should turn green.

After you enable the MQTT proxy, MQTT messages appear on the `freertos/demos/echo` topic, and data is printed to the UART terminal.

---

**Wi-Fi Provisioning**

Wi-Fi Provisioning is an Amazon FreeRTOS BLE service that allows you to securely send Wi-Fi network credentials from a mobile device to a microcontroller over BLE. The source code for the Wi-Fi Provisioning service can be found at `lib/bluetooth_low_energy/services/wifi_provisioning`.

*Note*

The Wi-Fi provisioning demo is currently supported on the Espressif ESP32-DevKitC. The Android version of the demo mobile application does not currently support Wi-Fi Provisioning.

**To enable the demo**

1. Confirm that the Wi-Fi Provisioning service is enabled in the `lib\utils\aws_ble_services_init.c` file. The service is enabled by default.
2. Configure the **Network Manager (p. 159)** to enable both BLE and Wi-Fi.

**To run the demo**

1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your microcontroller and your mobile device using the Amazon FreeRTOS BLE Mobile SDK Demo Application (p. 159).
3. From the **Devices** list in the demo mobile app, choose your microcontroller, and then choose **Network Config** to open the network configuration settings.
4. After you choose Network Config for your board, the microcontroller sends a list of the networks in the vicinity to the mobile device. Available Wi-Fi networks appear in a list under Scanned Networks. From the Scanned Networks list, choose your network, and then enter the SSID and password, if required.

The microcontroller connects to and saves the network. The network appears under the Saved Networks.
You can save several networks in the demo mobile app. When you restart the application and demo, the microcontroller connects to the first available saved network, starting from the top of the **Saved Networks** list.

To change the network priority order or delete networks, on the **Network Configuration** page, choose **Editing Mode**. To change the network priority order, choose the right side of the network that you want to reprioritize, and drag the network up or down. To delete a network, choose the red button on the left side of the network that you want to delete.

---

**Generic Attributes Server**

In this example, a demo Generic Attributes (GATT) server application on your microcontroller sends a simple counter value to the Amazon FreeRTOS BLE Mobile SDK Demo Application (p. 159) that is used for **MQTT over BLE** (p. 160) and **Wi-Fi Provisioning** (p. 162).

Using the BLE Mobile SDKs, you can create your own GATT client for a mobile device that connects to the GATT server on your microcontroller and runs in parallel with the demo mobile application.

**To run the demo**

1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your board and your mobile device using the Amazon FreeRTOS BLE Mobile SDK Demo Application (p. 159).
3. From the **Devices** list in the mobile SDK app, choose your board, and then choose **Custom GATT MQTT** to open the custom GATT service options.
4. Touch **Enable MQTT proxy** to enable the MQTT proxy. The slider should turn green.

5. Choose **Start Counter** to start publishing data to the `freertos/demos/echo` MQTT topic.

After you enable the MQTT proxy, Hello World and incrementing counter messages appear on the `freertos/demos/echo` topic.

## Secure Sockets Echo Client Demo

The following example uses a single RTOS task. The source code for this example can be found at `demos/common/tcp/aws_tcp_echo_client_single_task.c`.

Before you begin, verify that you have downloaded Amazon FreeRTOS to your microcontroller and built and run the Amazon FreeRTOS demo projects. You can download Amazon FreeRTOS from GitHub. For more information about setting up an Amazon FreeRTOS-qualified board, see Getting Started with Amazon FreeRTOS.

### To run the demo

1. Follow the instructions in the “TLS Server Setup” section of the Amazon FreeRTOS Qualification Program Developer Guide to set up a TLS Echo Server.

   By the end of step 6, the TLS Echo Server should be running and listening on the port 9000. You do not need to complete steps 7, 8, and 9.

   During the setup, you should have generated four files:
   - `client.pem` (client certificate)
   - `client.key` (client private key)
   - `server.pem` (server certificate)
   - `server.key` (server private key)

2. Use the tool `tools\certificate_configuration\CertificateConfigurator.html` to copy the client certificate (`client.pem`) and client private key (`client.key`) to `aws_clientcredential_keys.h`.

3. Open the `FreeRTOSConfig.h` file.
4. Set the `configECHO_SERVER_ADDR0`, `configECHO_SERVER_ADDR1`, `configECHO_SERVER_ADDR2`, and `configECHO_SERVER_ADDR3` variables to the four integers that make up the IP address where the TLS Echo Server is running.

5. Set the `configTCP_ECHO_CLIENT_PORT` variable to 9000, the port where the TLS Echo Server is listening.

6. Set the `configTCP_ECHO_TASKS_SINGLE_TASK_TLS_ENABLED` variable to 1.

7. Use the tool `tools\certificate_configuration\PEMfileToCString.html` to copy the server certificate (`server.pem`) to `cTlsECHO_SERVER_CERTIFICATE_PEM` in the file `aws_tcp_echo_client_single_task.c`.

8. In `demos/common/demo_runneraws_demo_runner.c`, switch the demo function to `vStartTCPPEchoClientTasks_SingleTasks()`:

```c
//extern void vStartMQTTEchoDemo( void );
extern void *vStartTCPPEchoClientTasks_SingleTasks*( void );

/**
 * @brief Runs demos in the system.
 */
void DEMO_RUNNER_RunDemos( void )
{
    //vStartMQTTEchoDemo();
    vStartTCPPEchoClientTasks_SingleTasks();
}
```

The microcontroller and the TLS Echo Server should be on the same network. When the demo starts (`main.c`), you should see a log message that reads `Received correct string from echo server`.

---

### Device Shadow Demo Application

The device shadow example demonstrates how to programmatically update and respond to changes in a device shadow. The device in this scenario is a light bulb whose color can be set to red or green. The device shadow example app is located in the `AmazonFreeRTOS/demos/common/shadow` directory. This example creates three tasks:

- A main demo task that calls `prvShadowMainTask`.
- A device update task that calls `prvUpdateTask`.
- A number of shadow update tasks that call `prvShadowUpdateTasks`.

`prvShadowMainTask` initializes the device shadow client and initiates an MQTT connection to AWS IoT. It then creates the device update task. Finally, it creates shadow update tasks and then terminates. The `democonfigSHADOW_DEMO_NUM_TASKS` constant defined in `AmazonFreeRTOS/demos/common/shadow/aws_shadow_lightbulb_on_off.c` controls the number of shadow update tasks created.

`prvShadowUpdateTasks` generates an initial thing shadow document and updates the device shadow with the document. It then goes into an infinite loop that periodically updates the thing shadow’s desired state, requesting the light bulb change its color (from red to green to red).

`prvUpdateTask` responds to changes in the device shadow’s desired state. When the desired state changes, this task updates the reported state of the device shadow to reflect the new desired state.

1. Add the following policy to your device certificate:

```
{
```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": "iot:Connect",
        "Resource": "arn:aws:iot:us-west-2:123456789012:client/<yourClientId>"
    },
    {
        "Effect": "Allow",
        "Action": "iot:Subscribe",
    },
    {
        "Effect": "Allow",
        "Action": "iot:Receive",
    },
    {
        "Effect": "Allow",
        "Action": "iot:Publish",
    }
]

2. Uncomment the declaration of and call to `vStartShadowDemoTasks` in `aws_demo_runner.c`. This function creates a task that runs the `prvShadowTask` function.

You can use the AWS IoT console to view your device's shadow and confirm that its desired and reported states change periodically.

1. In the AWS IoT console, from the left navigation pane, choose **Manage**.
2. Under **Manage**, choose **Things**, and then choose the thing whose shadow you want to view.
3. On the thing detail page, from the left navigation pane, choose **Shadow** to display the thing shadow.

For more information about how devices and shadows interact, see **Device Shadow Data Flow**.

#### Greengrass Discovery Demo Application

Before you run the FreeRTOS Greengrass Discovery demo, you must create a Greengrass group and then add a Greengrass core. For more information, see **Getting Started with AWS Greengrass**.

After you have a core running the Greengrass software, create an AWS IoT thing, certificate, and policy for your Amazon FreeRTOS device. For more information, see **Registering Your MCU Board with AWS IoT** (p. 5).

After you have created an IoT thing for your Amazon FreeRTOS device, follow the instructions to set up your environment and build Amazon FreeRTOS on one of the supported devices:

**Note**

Use the **Registering Your MCU Board with AWS IoT** (p. 5) instructions, but instead of downloading one of the predefined Connect to AWS IoT- XX configurations (where XX is TI, ST, NXP, Microchip, or Windows), download one of the Connect to AWS IoT Greengrass - XX configurations (where XX is TI, ST, NXP, Microchip, or Windows). Follow the steps in "Configure Your Project." Return to this topic after you have built Amazon FreeRTOS for your device.
At this point, you have downloaded the Amazon FreeRTOS software, imported it into your IDE, and built the project without errors. The project is already configured to run the Greengrass Connectivity demo. In the AWS IoT console, choose Test, and then add a subscription to freertos/demos/ggd. The demo publishes a series of messages to the Greengrass core. The messages are also published to AWS IoT, where they are received by the AWS IoT MQTT client.

In the MQTT client, you should see the following strings:

| Message from Thing to Greengrass Core: Hello world msg #1! |
| Message from Thing to Greengrass Core: Hello world msg #0! |
| Message from Thing to Greengrass Core: Address of Greengrass Core found!  <123456789012>.<us-west-2>.compute.amazonaws.com |

OTA Demo Application

Amazon FreeRTOS includes a demo application that demonstrates the use of the OTA library. The OTA demo application is located in the demos\common\ota subdirectory.

Before you create an OTA update, read Amazon FreeRTOS Over-the-Air Updates (p. 108) and complete all prerequisites listed there.

The OTA demo application:

1. Initializes the FreeRTOS network stack and MQTT buffer pool. (See main.c.)
2. Creates a task to exercise the OTA library. (See vOTAUpdateDemoTask in aws_ota_update_demo.c.)
3. Creates an MQTT client using MQTT_AGENT_Create.
4. Connects to the AWS IoT MQTT broker using MQTT_AGENT_Connect.
5. Calls OTA_AgentInit to create the OTA task and registers a callback to be used when the OTA task is complete.

You can use the AWS IoT console or the AWS CLI to create an OTA update job. After you have created an OTA update job, connect a terminal emulator to see the progress of the OTA update. Make a note of any errors generated during the process.

A successful OTA update job displays output like the following. Some lines in this example have been removed from the listing for brevity.

```
313 267848 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
314 268733 [OTA Task] [OTA] Set job doc parameter [jobId: fe1ec7e2_8c31_4438_b0b9_ad55adc95610]
315 268734 [OTA Task] [OTA] Set job doc parameter [streamname: 327]
316 268734 [OTA Task] [OTA] Set job doc parameter [filepath: /sys/mcuflashimg.bin]
```
OTA Demo Application

317 268734 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
318 268735 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
319 268735 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
320 268735 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
321 268737 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa: 56qgHQ3Lxv6Kkorv1lV4A4yGWBmS3d ]
322 268737 [OTA Task] [OTA] Job was accepted. Attempting to start transfer.
323 268737 [OTA Task] [OTA] Sending command to MQTT task.
324 268737 [MQTT] Received message 50000 from queue.
325 268848 [OTA] [OTA] Queued: 2   Processed: 1   Dropped: 0
326 269039 [MQTT] MQTT Subscribe was accepted. Subscribed.
327 269039 [MQTT] Notifying task.
328 269040 [OTA Task] Command sent to MQTT task passed.
329 269041 [OTA Task] Subscribed to topic: $aws/things/TI-LaunchPad/streams/327
330 269848 [OTA] [OTA] Queued: 2   Processed: 1   Dropped: 0
... // Output removed for brevity
346 269409 [OTA Task] [OTA] file token: 74594452
... // Output removed for brevity
363 301327 [OTA Task] [OTA] file ready for access.
364 301327 [OTA Task] [OTA] Returned buffer to MQTT Client.
365 301328 [OTA Task] Sending command to MQTT task.
366 301328 [MQTT] Received message 60000 from queue.
367 301328 [MQTT] Notifying task.
368 301329 [OTA Task] Command sent to MQTT task passed.
369 301329 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/streams/327/get
370 301647 [OTA Task] [OTA] Remaining: 127
... // Output removed for brevity
508 304622 [OTA Task] Sending command to MQTT task.
509 304622 [MQTT] Received message 70000 from queue.
510 304622 [MQTT] Notifying task.
511 304623 [OTA Task] Command sent to MQTT task passed.
512 304623 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/streams/327/get
513 304860 [OTA] [OTA] Queued: 47   Processed: 47   Dropped: 83
514 304926 [OTA Task] [OTA] Received file block 4, size 1024
515 304941 [OTA Task] [OTA] Remaining: 82
... // Output removed for brevity
797 315047 [MQTT] MQTT Publish was successful.
798 315048 [MQTT] Notifying task.
799 315048 [OTA Task] Command sent to MQTT task passed.
800 315049 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/jobs/e18c7ec_8c31_4438_a2b9_9d5ac69567801 315049 [OTA Task] Sending command to MQTT task.
802 315049 [MQTT] Received message d0000 from queue.
803 315150 [MQTT] MQTT Unsubscribe was successful.
804 315150 [MQTT] Notifying task.
805 315151 [OTA Task] Command sent to MQTT task passed.
806 315152 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327
807 315172 [OTA Task] Sending command to MQTT task.
808 315172 [MQTT] Received message e0000 from queue.
809 315273 [MQTT] MQTT Unsubscribe was successful.
810 315273 [MQTT] Notifying task.
811 315274 [OTA Task] Command sent to MQTT task passed.
812 315274 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327
813 315275 [OTA Task] [OTA] Resetting MCU to activate new image.
0 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created

Device came up in Station mode

2 137 [Tmr Svc] Wi-Fi module initialized.
3 137 [Tmr Svc] Starting key provisioning...

169
4 137 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 339 [Tmr Svc] Write device certificate...
7 436 [Tmr Svc] Key provisioning done...
Device disconnected from the AP on an ERROR.!!


[NETAPP EVENT] IP acquired by the device

Device has connected to Guest
Device IP Address is 192.168.3.72

8 1443 [Tmr Svc] Wi-Fi connected to AP Guest.
9 1444 [Tmr Svc] IP Address acquired 192.168.3.72
10 1444 [OTA] OTA demo version 0.9.1
11 1445 [OTA] Creating MQTT Client...
12 1445 [OTA] Connecting to broker...
13 1445 [OTA] Sending command to MQTT task.
14 1445 [MQTT] Received message 10000 from queue.
15 2910 [MQTT] MQTT Connect was accepted. Connection established.
16 2910 [MQTT] Notifying task.
17 2911 [OTA] Command sent to MQTT task passed.
18 2912 [OTA] Connected to broker.
19 2913 [OTA Task] Sending command to MQTT task.
20 2913 [MQTT] Received message 20000 from queue.
21 3014 [MQTT] MQTT Subscribe was accepted. Subscribed.
22 3014 [MQTT] Notifying task.
23 3015 [OTA Task] Command sent to MQTT task passed.
24 3015 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/$next/get/accepted

25 3028 [OTA Task] Sending command to MQTT task.
26 3028 [MQTT] Received message 30000 from queue.
27 3129 [MQTT] MQTT Subscribe was accepted. Subscribed.
28 3129 [MQTT] Notifying task.
29 3130 [OTA Task] Command sent to MQTT task passed.
30 3138 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/notify-next

31 3138 [OTA Task] [OTA] Check For Update #0
32 3138 [OTA Task] Sending command to MQTT task.
33 3138 [MQTT] Received message 40000 from queue.
34 3241 [MQTT] MQTT Publish was successful.
35 3241 [MQTT] Notifying task.
36 3243 [OTA Task] Command sent to MQTT task passed.
37 3245 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:TI-LaunchPad ]
38 3245 [OTA Task] [OTA] Set job doc parameter [ jobId: fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
39 3245 [OTA] [OTA] Identified job doc parameter [ self_test ]
40 3246 [OTA Task] [OTA] Set job doc parameter [ updatedBy: 589827 ]
41 3246 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
42 3246 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
43 3247 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
44 3247 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
45 3247 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
46 3247 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
47 3247 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa: Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
48 3249 [OTA Task] [OTA] Job is ready for self test.
49 3250 [OTA Task] [OTA] Sending command to MQTT task.
50 3351 [MQTT] MQTT Publish was successful.
51 3352 [MQTT] Notifying task.
52 3352 [OTA Task] Command sent to MQTT task passed.
Demo Bootloader for the Microchip Curiosity PIC32MZEF

This demo bootloader implements firmware version checking, cryptographic signature verification, and application self-testing. These capabilities support over-the-air (OTA) firmware updates for Amazon FreeRTOS.

The firmware verification includes verifying the authenticity and integrity of the new firmware received over the air. The bootloader verifies the cryptographic signature of the application before booting. The demo uses elliptic-curve digital signature algorithm (ECDSA) over SHA256. The utilities provided can be used to generate a signed application that can be flashed on the device.

The bootloader supports the following features required for OTA:

- Maintains application images on the device and switches between them.
- Allows self-test execution of a received OTA image and roll-back on failure.
- Checks signature and version of the OTA update image.

Bootloader States

The bootloader process is described by the following state machine.
The following table describes the bootloader states.

<table>
<thead>
<tr>
<th><strong>Bootloader State</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Bootloader is in the initialization state.</td>
</tr>
<tr>
<td>Verification</td>
<td>Bootloader is verifying the images present on the device.</td>
</tr>
<tr>
<td>Execute Image</td>
<td>Bootloader is launching the selected image.</td>
</tr>
<tr>
<td>Execute Default</td>
<td>Bootloader is launching the default image.</td>
</tr>
<tr>
<td>Error</td>
<td>Bootloader is in the error state.</td>
</tr>
</tbody>
</table>

In the preceding diagram, both **Execute Image** and **Execute Default** are shown as the **Execution state**.

**Bootloader Execution State**

The bootloader is in the **Execution** state and is ready to launch the selected verified image. If the image to be launched is in the upper bank, the banks are swapped before executing the image, because the application is always built for the lower bank.

**Bootloader Default Execution State**

If the configuration option to launch the default image is enabled, the bootloader launches the application from a default execution address. This option must be disabled except while debugging.

**Bootloader Error State**

The bootloader is in an error state and no valid images are present on the device. The bootloader must notify the user. The default implementation sends a log message to the console and fast-blinks the LED on the board indefinitely.

**Flash Device**

The Microchip Curiosity PIC32MZEF platform contains an internal program flash of two megabytes divided into two banks. It supports memory map swapping between these two banks and live updates. The demo bootloader is programmed in a separate lower boot flash region.
Application Image Structure

The diagram shows the main components of the application image stored on each bank of the device.

<table>
<thead>
<tr>
<th>Component</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image header</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Image descriptor</td>
<td>24 bytes</td>
</tr>
<tr>
<td>Application binary</td>
<td>&lt; 1 MB - (324)</td>
</tr>
<tr>
<td>Trailer</td>
<td>292 bytes</td>
</tr>
</tbody>
</table>

Image Header

The application images present on the device must start with a header that consists of a magic code and image flags.

<table>
<thead>
<tr>
<th>Header Field</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic Code</td>
<td>7 bytes</td>
</tr>
<tr>
<td>Image Flags</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

Magic Code

The image on the flash device must start with a magic code. The default magic code is @AFRTOS. The bootloader checks if a valid magic code is present before booting the image. This is the first step of verification.
Image Flags

The image flags are used to store the status of the application images. The flags are used in the OTA process. The image flags of both banks determine the state of the device. If the executing image is marked as commit pending, it means the device is in the OTA self-test phase. Even if images on the devices are marked valid, they go through the same verification steps on every boot. If an image is marked as new, the bootloader marks it as commit pending and launches it for self-test after verification. The bootloader also initializes and starts the watchdog timer so that if the new OTA image fails self-test, the device reboots and bootloader rejects the image by erasing it and executes the previous valid image.

The device can have only one valid image. The other image can be a new OTA image or a commit pending (self-test). After a successful OTA update, the old image is erased from the device.

<table>
<thead>
<tr>
<th>Status</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New image</td>
<td>0xFF</td>
<td>Application image is new and never executed.</td>
</tr>
<tr>
<td>Commit pending</td>
<td>0xFE</td>
<td>Application image is marked for test execution.</td>
</tr>
<tr>
<td>Valid</td>
<td>0xFC</td>
<td>Application image is marked valid and committed.</td>
</tr>
<tr>
<td>Invalid</td>
<td>0xF8</td>
<td>Application image is marked invalid.</td>
</tr>
</tbody>
</table>

Image Descriptor

The application image on the flash device must contain the image descriptor following the image header. The image descriptor is generated by a post-build utility that uses configuration files (ota-descriptor.config) to generate the appropriate descriptor and prepends it to the application binary. The output of this post-build step is the binary image that can be used for OTA.

<table>
<thead>
<tr>
<th>Descriptor Field</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Start Address</td>
<td>4 bytes</td>
</tr>
<tr>
<td>End Address</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Execution Address</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Hardware ID</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Reserved</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Sequence Number

The sequence number must be incremented before building a new OTA image. See the ota-descriptor.config file. The bootloader uses this number to determine the image to boot. Valid values are from 1 to 4294967295.
Start Address

The starting address of the application image on the device. As the image descriptor is prepended to the application binary, this address is the start of the image descriptor.

End Address

The ending address of the application image on the device, excluding the image trailer.

Execution Address

The execution address of the image.

Hardware ID

A unique hardware ID used by the bootloader to verify the OTA image is built for the correct platform.

Reserved

This is reserved for future use.

Image Trailer

The image trailer is appended to the application binary. It contains the signature type string, signature size, and signature of the image.

<table>
<thead>
<tr>
<th>Trailer Field</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature Type</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Signature Size</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Signature</td>
<td>256 bytes</td>
</tr>
</tbody>
</table>

Signature Type

The signature type is a string that represents the cryptographic algorithm being used and serves as a marker for the trailer. The bootloader supports the elliptic-curve digital signature algorithm (ECDSA). The default is sig-sha256-ecdsa.

Signature Size

The size of the cryptographic signature, in bytes.

Signature

The cryptographic signature of the application binary prepended with the image descriptor.

Bootloader Configuration

The basic bootloader configuration options are provided in aws_boot_config.h. Some options are provided for debugging purposes only. aws_boot_config.h is located in /demos/microchip/curiosity_pic32_bl/config_files/.

Enable Default Start

Enables the execution of the application from the default address and must be enabled for debugging only. The image is executed from the default address without any verification.
Enable Crypto Signature Verification

Enables cryptographic signature verification on boot. Failed images are erased from the device. This option is provided for debugging purposes only and must remain enabled in production.

Erase Invalid Image

Enables a full bank erase if image verification on that bank fails. The option is provided for debugging and must remain enabled in production.

Enable Hardware ID Verification

Enables verification of the hardware ID in the descriptor of the OTA image and the hardware ID programmed in the bootloader. This is optional and can be disabled if hardware ID verification is not required.

Enable Address Verification

Enables verification of the start, end, and execution addresses in the descriptor of OTA image. We recommend that you keep this option enabled.

Building the Bootloader

The demo bootloader is included as a loadable project in the aws_demos project located under demos\microchip\curiosity_pic32mzef\mplab in the Amazon FreeRTOS source code repository. When the aws_demos project is built, it builds the bootloader first, followed by the application. The final output is a unified hex image including the bootloader and the application. The factory_image_generator.py utility is provided to generate a unified hex image with cryptographic signature. The bootloader utility scripts are located in /demos/common/ota/bootloader/utility/.

Bootloader Pre-Build Step

This pre-build step executes a utility script called codesigner_cert_utility.py that extracts the public key from the code-signing certificate and generates a C header file that contains the public key in ASN.1 encoded format. This header is compiled into the bootloader project. The generated header contains two constants: an array of the public key and the length of the key. The bootloader project can also be built without aws_demos and can be debugged as normal application.
Troubleshooting Amazon FreeRTOS

Amazon FreeRTOS supports Amazon CloudWatch and AWS CloudTrail logging services to help troubleshoot issues with Amazon FreeRTOS Over-the-Air updates. For more information about troubleshooting OTA updates, see OTA Troubleshooting.
Amazon FreeRTOS Porting Guide

This porting guide walks you through modifying the Amazon FreeRTOS software package to work on boards that are not Amazon FreeRTOS qualified. Amazon FreeRTOS is designed to let you choose only those libraries required by your board or application. The MQTT, Shadow, and Greengrass libraries are designed to be compatible with most devices as-is, so there is no porting guide for these libraries.

For information about porting FreeRTOS kernel, see FreeRTOS Kernel Porting Guide.

Topics
- Bootloader (p. 178)
- Logging (p. 178)
- Connectivity (p. 179)
- Security (p. 180)
- Using Custom Libraries with Amazon FreeRTOS (p. 182)
- OTA Portable Abstraction Layer (p. 182)

Bootloader

The bootloader must be dual-bank capable and include logic for checking a CRC and app version in the image header. The bootloader boots the newest image, based on the app version in the header, provided that the CRC is valid. If the CRC check fails, the bootloader should zero out the header as an optimization for future reboots.

Since the OTA v1 agent performs cryptographic signature verification, we suggest that v1 bootloaders not link to cryptographic code, so as to be as small as possible. You must provide a compliant bootloader.

Logging

Amazon FreeRTOS provides a thread-safe logging task that can be used by calling the configPRINTF function. configPRINTF is designed to behave like printf. To port configPRINTF, initialize your communications peripheral, and define the configPRINT_STRING macro so that it takes an input string and displays it on your preferred output.

Logging Configuration

configPRINT_STRING should be defined for your board’s implementation of logging. Current examples use a UART serial terminal, but other interfaces can also be used.

#define configPRINT_STRING( x )

Use configLOGGING_MAX_MESSAGE_LENGTH to set the maximum number of bytes to be printed. Messages longer than this length are truncated.

#define configLOGGING_MAX_MESSAGE_LENGTH
When `configLOGGING_INCLUDE_TIME_AND_TASK_NAME` is set to 1, all printed messages are prepended with additional debug information (the message number, FreeRTOS tick count, and task name).

```
#define configLOGGING_INCLUDE_TIME_AND_TASK_NAME 1
```

`vLoggingPrintf` is the name of the FreeRTOS thread-safe printf call. You do not need to change this value to use AmazonFreeRTOS logging.

```
#define configPRINTF( X ) vLoggingPrintf X
```

## Connectivity

You must first configure your connectivity peripheral. You can use Wi-Fi, Bluetooth, Ethernet, or other connectivity mediums. At this time, only a Wi-Fi management API is defined for board ports, but if you are using Ethernet, the FreeRTOS TCP/IP software can provide a good place to start.

### Wi-Fi Management

The Wi-Fi management library supports network connectivity following the 802.11 (a/b/n) protocol. If your hardware does not support Wi-Fi, you do not need to port this library.

The functions that must be ported are listed in the `lib/wifi/portable/<vendor>/<platform>/aws_wifi.c` file. You can find a detailed explanation for each public interface in `lib/include/aws_wifi.h`.

The following functions must be ported:

```c
WiFiReturnCode_t WIFI_On( void );
WiFiReturnCode_t WIFI_Off( void );
WiFiReturnCode_t WIFI_ConnectAP( const WiFiNetworkParams_t * const pxNetworkParams );
WiFiReturnCode_t WIFI_Disconnect( void );
WiFiReturnCode_t WIFI_Reset( void );
WiFiReturnCode_t WIFI_Scan( WiFiScanResult_t * pxBuffer, uint8_t uxNumNetworks );
```

### Sockets

The sockets library supports TCP/IP network communication between your board and another node in the network. The sockets APIs are based on the Berkeley sockets interface, but also include a secure communication option. At this time, only client APIs are supported. We recommend that you port the TCP/IP functionality first, before you add the TLS functionality.

Libraries for MQTT, Shadow, and Greengrass all make calls into the sockets layer. A successful port of the sockets layer allows the protocols built on sockets to just work.

### Major Differences from Berkeley Sockets Implementation

#### Security

The sockets interface must be configured to use TLS for secure communication. The `SetSockOpt` command has a couple of nonstandard options that must be implemented to work with AmazonFreeRTOS examples.
Security

Amazon FreeRTOS has two libraries that work together to provide platform security: TLS and PKCS#11. Amazon FreeRTOS provides a software security solution built on mbed TLS (a third-party TLS library). The TLS API uses mbed TLS to encrypt and authenticate network traffic. PKCS#11 provides an standard interface to handle cryptographic material and replace software cryptographic operations with implementations that fully use the hardware.

TLS

If you choose to use an mbed TLS-based implementation, you can use aws_tls.c as-is, provided that PKCS#11 is implemented.

The public interfaces of this library and a detailed explanation for each TLS interface are listed in lib/include/aws_tls.h. The Amazon FreeRTOS implementation of the TLS library is in lib/tls/aws_tls.c. If you decide to use your own TLS support, you can either implement the TLS public interfaces and plug them into the sockets public interfaces, or you can directly port the sockets library using your own TLS interfaces.

The mbedtls_hardware_poll function provides randomness for the deterministic random bit generator. For security, no two boards should provide identical randomness, and a board must not provide the same random value repeatedly, even if the board is reset. Examples of implementations for this function can be found in ports using mbed TLS at demos/<vendor>/<platform>/common/application_code/<vendor code>/aws_entropy_hardware_poll.c

Using TLS Libraries Other Than mbed TLS

If you are porting another TLS library to Amazon FreeRTOS, make sure that a compatible TLS cipher suite is implemented in your port. For more information, see Cipher Suites Compatible with AWS IoT. The following cipher suites are compatible with AWS IoT Greengrass devices:

- **TLS_RSA_WITH_AES_128_GCM_SHA256**
PKCS#11

Amazon FreeRTOS implements a PKCS#11 standard for cryptographic operations and key storage. The header file for PKCS#11 is an industry standard. To port PKCS#11, you must implement functions to read and write credentials to and from non-volatile memory (NVM).

The functions you need to implement are listed in lib/third_party/pkcs11/pkcs11f.h. The implementation of the public interfaces is located in lib/pkcs11/portable/vendor/board/pkcs11.c.

The following functions are the minimum required to support TLS client authentication in Amazon FreeRTOS:

- C_GetFunctionList
- C_Initialize
- C_GetSlotList
- C_OpenSession
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_GetAttributeValue
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_GetAttributeValue
- C_SignInit
- C_Sign
- C_CloseSession
- C_Finalize

For a general porting guide, see the open standard, PKCS #11 Cryptographic Token Interface Base Specification.

Two additional non-PKCS#11 standard functions must be implemented for keys and certificates to survive power cycle:

prvSaveFile

Writes the client (device) private key and certificate to memory. If your NVM is susceptible to damage from write cycles, you might want to use an additional variable to record whether the device private key and device certificate have been initialized.
prvReadFile

Retrieves either the device private key or device certificate from NVM into RAM for use by the TLS library.

Using Custom Libraries with Amazon FreeRTOS

All Amazon FreeRTOS libraries can be replaced with custom developed libraries. All custom libraries must conform to the API of the Amazon FreeRTOS library they replace.

OTA Portable Abstraction Layer

Amazon FreeRTOS defines an OTA portable abstraction layer (PAL) in order to ensure that the OTA library is useful on a wide variety of hardware. The OTA PAL interface is listed below.

prvAbort

Aborts an OTA update.

prvCreateFileForRx

Creates a new file to store the data chunks as they are received.

prvCloseFile

Closes the specified file. This may authenticate the file if it is marked as secure.

prvCheckFileSignature

Verifies the signature of the specified file. For device file systems with built-in signature verification enforcement, this may be redundant and should therefore be implemented as a no-op.

prvWriteBlock

Writes a block of data to the specified file at the given offset. Returns the number of bytes written on success or negative error code.

prvActivateNewImage

Activates the new firmware image. For some ports, this function may not return.

prvSetImageState

Does whatever is required by the platform to accept or reject the last firmware image (or bundle). Refer to the platform implementation to determine what happens on your platform.

prvReadAndAssumeCertificate

Reads the specified signer certificate from the file system and returns it to the caller. This is optional on some platforms.
The Amazon FreeRTOS Qualification Program is now a part of the Device Qualification Program. For more information about the Device Qualification Program, visit the AWS Partner Network website.
AWS IoT Device Tester for Amazon FreeRTOS User Guide

AWS IoT Device Tester allows you to test that the Amazon FreeRTOS operating system works locally on your device and can communicate with the AWS IoT cloud. AWS IoT Device Tester checks if the porting layer interfaces for Amazon FreeRTOS libraries function correctly on top of microcontroller board device drivers. In addition, it performs end-to-end tests with AWS IoT Core (for example, to test if the board is able to send or receive MQTT messages and process correctly). AWS IoT Device Tester for Amazon FreeRTOS uses the test cases published in the Amazon FreeRTOS GitHub repository. AWS IoT Device Tester consists of a Test Manager command-line tool and a set of test cases.

Test Manager runs on a host computer (Windows, Mac, or Linux) that is connected to the device to be tested. The Test Manager executes test cases and aggregates results. It also provides a command line interface to manage test execution. Test cases contain test logic and set up the resources required for tests.

Test cases are part of the application binary image that is flashed onto your board. Application binary images include Amazon FreeRTOS, the semiconductor vendor’s ported Amazon FreeRTOS interfaces, and board device drivers. Test Cases run as embedded applications and verify the ported Amazon FreeRTOS interfaces function correctly on top of the device drivers.

The following diagram shows the test infrastructure setup:

Prerequisites

This section describes the prerequisites for testing microcontrollers with AWS IoT Device Tester.

Download Amazon FreeRTOS

You can download the version of Amazon FreeRTOS that you want to test from GitHub. If you are using Windows, you must keep the file path short. For example, to avoid a Windows limitation with
long file paths, clone to C:\AFreeRTOS rather than C:\Users\username\programs\projects AmazonFreeRTOS\.

Download AWS IoT Device Tester for Amazon FreeRTOS

Every version of Amazon FreeRTOS has a corresponding version of AWS IoT Device Tester for performing qualification tests. Download the appropriate version of AWS IoT Device Tester.

Extract AWS IoT Device Tester into a location on the file system where you have read and write permissions. Due to a path length limitation, on Microsoft Windows, extract AWS IoT Device Tester into a root directory like C:\ or D:\.

Create and Configure an AWS Account

If you don’t have an AWS account, follow the instructions on the AWS webpage to create one. Choose Create an AWS Account and follow the prompts.

Create an IAM User in Your AWS Account

When you create an AWS account, a root user that has access to all resources in your account is created for you. It is a best practice to create another user for everyday tasks. To create an IAM user, follow the instructions in Creating an IAM User in Your AWS Account. For more information about the root user, see The AWS Account Root User.

Create and Attach an IAM Policy to Your AWS Account

IAM policies grant your IAM user access to AWS resources.

To create an IAM policy

1. Browse to the IAM console.
2. In the navigation pane, choose Policies, and then choose Create Policy.
3. Select the JSON tab and copy and paste the policy template located in Permissions Policy Template (p. 202) the ?? (p. 202) into the editor window.
5. In Name, enter a name for your policy. In Description, enter an optional description. Choose Create Policy.

After you create an IAM policy, you must attach it to your IAM user.

To attach an IAM policy to your IAM user

1. Browse to the IAM console.
2. In the navigation pane, choose Users. Find and select your IAM user.
3. Choose Add permissions, and then choose Attach existing policies directly. Find and select your IAM policy, choose Next: Review, and then choose Add Permissions.

Install the AWS Command Line Interface (CLI)

You will need to use the CLI to perform some operations, if you don't have the CLI installed, follow the instructions in Install the AWS CLI.
Test to Qualify Your Microcontroller Board for the First Time

You can use AWS IoT Device Tester to test as you port the Amazon FreeRTOS interfaces. After you have ported the Amazon FreeRTOS interfaces for your board’s device drivers, you use AWS IoT Device Tester to run the qualification tests on your microcontroller board.

Add Library Porting Layers

To add library porting layers for Amazon FreeRTOS device libraries (TCP/IP, WiFi, and so on) compatible with your MCU architecture, you must:

1. Implement the configPRINT_STRING() method before running AWS IoT Device Tester tests. AWS IoT Device Tester calls the configPRINT_STRING() macro to output test results as human-readable ASCII strings.
2. Port the drivers to implement the Amazon FreeRTOS library’s interfaces. For more information, see the Amazon FreeRTOS Qualification Developer Guide.

Configure Your AWS Credentials

You must configure your AWS credentials in the <devicetester_extract_location>/devicetester_afreertos_[win/mac/linux]/configs/config.json. You can specify your credentials in one of two ways:

- Environment variables
- Credentials file

Configuring AWS Credentials with Environment Variables

Environment variables are variables maintained by the operating system and used by system commands. AWS IoT Device Tester can use the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables to store your AWS credentials. The way you set environment variables depends on the operating system you are running.

To set environment variables on Windows

1. From the Windows 10 desktop, open the Power User Task Menu. To open the menu, move your mouse cursor to the bottom-left corner of the screen (the Start menu icon) and right-click.
2. From the Power User Task Menu, choose System, and then choose Advanced System Settings. Note
   In Windows 10, you might need to scroll to Related settings and choose System info. In System, choose Advanced system settings.
   In System Properties, choose the Advanced tab, and then choose the Environment Variables button.
3. Under User variables for <user-name>, choose New to create an environment variable. Enter a name and value for each environment variable.

To set environment variables on macOS, Linux, or UNIX

- Open ~/.bash_profile in any text editor and add the following lines:
Create a Device Pool in AWS IoT Device Tester

Devices to be tested are organized in device pools. Each device pool consists of one or more devices with identical specifications. You can configure Device Tester to test a single device in a pool or multiple devices in a pool. To accelerate the qualification process, AWS IoT Device Tester can test devices with the same specification in parallel. It uses a round-robin method to execute a different test group on each device in a device pool.

Configuring AWS IoT Device Tester for Single Device Testing

You define a device pool by editing the `device.json` file template in the configs folder. The following is an example `device.json` file used to create a device pool with one device:
The following list describes the attributes used in the `device.json` file:

**id**

A user-defined alphanumeric ID that uniquely identifies a pool of devices. Devices that belong to a pool must be of the same type. When a suite of tests is running, devices in the pool are used to parallelize the workload.

**sku**

An alphanumeric value that uniquely identifies the board you are testing. The SKU is used to track qualified boards.

**Note**

If you want to list your board in AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

**features**

An array that contains the device's supported features. The Device Tester uses this information to select the qualification tests to run.

Supported values are:

- **TCP/IP**: Indicates if your board supports a TCP/IP stack and whether it is supported on-chip (MCU) or offloaded to another module.
• **WIFI**: Indicates if your board has Wi-Fi capabilities.
• **TLS**: Indicates if your board supports TLS and if it is supported on-chip (MCU) or offloaded to another module.
• **OTA**: Indicates if your board supports over-the-air (OTA) update functionality.

devices.id

A user-defined unique identifier for the device being tested.

devices.connectivity.protocol

The communication protocol used to communicate with this device. Supported value: *uart*.

devices.connectivity.serialPort

The serial port of the host computer used to connect to the devices being tested.

identifiers

Optional. An array of arbitrary name-value pairs. You can use these values in the build and flash commands described in the next section.

### Configuring AWS IoT Device Tester for Multiple Device Testing

You can add multiple devices by editing the `devices` section of the `device.json` template in the `configs` folder. For more information about the structure and contents of the `device.json` file, see Configuring AWS IoT Device Tester for Single Device Testing (p. 187).

**Note**

All devices in the same pool must be of same technical specification and SKU.

The following is an example `device.json` file used to create a device pool with multiple devices:

```json
[
  {
    "id": "<pool-id>",
    "sku": "<sku>",
    "features": [
      {
        "name": "WIFI",
        "value": "Yes | No"
      },
      {
        "name": "OTA",
        "value": "Yes | No"
      },
      {
        "name": "TCP/IP",
        "value": "On-chip | Offloaded | No"
      },
      {
        "name": "TLS",
        "value": "On-chip | Offloaded | No"
      }
    ],
    "devices": [
      {
        "id": "<device-id1>",
        "connectivity": {
          "protocol": "uart",
          "serialPort": "<computer_serial_port_1>"
        },
        "identifiers": [
          {
            "name": "serialNo",
            "value": "<serialNo-value>"
          }
        ]
      }
    ]
  }
]
```
Configure Build, Flash, and Test Settings

For AWS IoT Device Tester to build and flash test cases on to your board automatically, you must configure AWS IoT Device Tester with command line interfaces of your build and flash tools. The build and flash settings are configured in the userdata.json template file located in the config folder.

Configure Settings for Testing One Device

The userdata.json must have the following structure:

```json
{
  "sourcePath": "<path-to-afr-source-code>",
  "buildTool": {
    "name": "<your-build-tool-name>",
    "version": "<your-build-tool-version>",
    "command": ["<your-build-script>"
  }
},
  "flashTool": {
    "name": "<your-flash-tool-name>",
    "version": "<your-flash-tool-version>",
    "command": ["<your-flash-script>"
  }
},
  "clientWifiConfig": {
    "wifiSSID": "<your-wifi-ssid",
    "wifiPassword": "<your-wifi-password>",
    "wifiSecurityType": "<wifi-security-type>"
  },
  "testWifiConfig": {
    "wifiSSID": "<your-wifi-ssid",
    "wifiPassword": "<your-wifi-password>",
    "wifiSecurityType": "<wifi-security-type>"
  },
  "otaConfiguration": {
    "otaFirmwareFilePath": "<path-to-the-device-binary>",
    "deviceFirmwareFileName": "<your-device-firmware-name>.bin",
    "awsSignerPlatform": "AmazonFreeRTOS-Default",
    "awsUntrustedSignerCertificateArn": "arn:aws:acm:us-east-1:1000000001:certificate/0c81e2c6-f55e-46b1-9ed1-2c404309b210",
    "awsSignerCertificateFileName": "ecdsa-sha256-signer.crt.pem",
    "compileCodesignerCertificate": true
  }
}
```
The following lists the attributes used in the `userdata.json` file:

**sourcePath**

The path to the root of the ported Amazon FreeRTOS source code. AWS IoT Device Tester stores the value in the `{{testData.sourcePath}}` variable.

**buildTool**

The full path to your build script (.bat or .sh) that contains the commands to build your source code.

*Note*

If you are using an IDE, you must provide the command line to the IDE to run in headless mode.

**flashTool**

Full path to your flash script (.sh or .bat) that contains the flash commands for your device.

**clientWifiConfig**

Client Wi-Fi configuration. The Wi-Fi library tests require an MCU board to connect to two access points. This attribute configures the Wi-Fi settings for the first access point. The client Wi-Fi settings are configured in `$AFR_HOME/tests/common/include/aws_clientcredential.h`. The following macros are set by using the values found in `aws_clientcredential.h`. Some of the Wi-Fi test cases expect the access point to have some security and not to be open.

- `wifi_ssid`: The Wi-Fi SSID as a C string.
- `wifi_password`: The Wi-Fi password as a C string.
- `wifiSecurityType`: The type of Wi-Fi security used.

**testWifiConfig**

Test Wi-Fi configuration. The Wi-Fi library tests require a board to connect to two access points. This attribute configures the second access point. The test Wi-Fi settings are configured in `$AFR_HOME/tests/common/wifi/aws_test_wifi.c`. The following macros are set by using the values found in `aws_test_wifi.c`. Some of the Wi-Fi test cases expect the access point to have some security and not to be open.

*Note*

If your board does not support Wi-Fi, you must still include the `clientWifiConfig` and `testWifiConfig` section in your `device.json` file, but you can omit values for these attributes.

- `testwifiWIFI_SSID`: The Wi-Fi SSID as a C string.
- `testwifiWIFI_PASSWORD`: The Wi-Fi password as a C string.
- `testwifiWIFI_SECURITY`: The type of Wi-Fi security used. One of the following values:
  - `wifiSecurityOpen`
  - `wifiSecurityWEP`
  - `wifiSecurityWPA`
  - `wifiSecurityWPA2`

**otaConfiguration**

The OTA configuration.

**otaFirmwareFilePath**

The full path to the OTA image created after the build.

**deviceFirmwareFileName**

The full file path on the MCU device where the OTA firmware will be downloaded. Some devices do not use this field, but you still must provide a value.
awsSignerPlatform

The signing algorithm used by AWS Code Signer while creating the OTA update job. Currently, the possible values for this field are AmazonFreeRTOS-TI-CC3220SF and AmazonFreeRTOS-Default.

awsSignerCertificateArn

The Amazon Resource Name (ARN) for the trusted certificate uploaded to AWS Certificate Manager (ACM). For more information about creating a trusted certificate, see Creating a Code Signing Certificate.

awsUntrustedSignerCertificateArn

The Amazon Resource Name (ARN) for a certificate uploaded to ACM which your device should not trust. This is used to test invalid certificate test cases.

compileCodesignerCertificate

Set to true if the code-signer signature verification certificate is not provisioned or flashed, so it must be compiled into the project. AWS IoT Device Tester fetches the trusted certificate from ACM and compiles it into aws_codesigner_certificate.h.

Configure Settings for Testing Multiple Devices

Build, flash, and test settings are made in the userdata.json file. The following JSON example shows how you can configure AWS IoT Device Tester for testing multiple devices:

```json
{
    "sourcePath": "<absolute-path-to/amazon-freertos>",
    "buildTool": {
        "name": "<your-build-tool-name>",
        "version": "<your-build-tool-version>",
        "command": [
            "<absolute-path-to/build-parallel-script> {{testData.sourcePath}}"
        ]
    },
    "flashTool": {
        "name": "<your-flash-tool-name>",
        "version": "<your-flash-tool-version>",
        "command": [
            "<absolute-path-to/flash-parallel-script> {{testData.sourcePath}} {{device.connectivity.serialPort}}"
        ]
    },
    "clientWifiConfig": {
        "wifiSSID": "<ssid>",
        "wifiPassword": "<password>",
        "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA | eWiFiSecurityWPA2"
    },
    "testWifiConfig": {
        "wifiSSID": "<ssid>",
        "wifiPassword": "<password>",
        "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA | eWiFiSecurityWPA2"
    },
    "otaConfiguration": {
        "otaFirmwareFilePath": "{{testData.sourcePath}}/{{relative-path-to/ota-image-generated-in-build-process}}",
        "deviceFirmwareFileName": "<absolute-path-to/ota-image-on-device>",
        "awsSignerPlatform": "AmazonFreeRTOS-Default | AmazonFreeRTOS-TI-CC3220SF",
    }
}
```
"awsSignerCertificateFileName": "<awsSignerCertificate-file-name>",
"compileCodesignerCertificate": true | false
}
}
}

The following lists the attributes used in `userdata.json`:

sourcePath
The path to the root of the ported Amazon FreeRTOS source code. AWS IoT Device Tester stores the value in the `{ testData.sourcePath }` variable.

buildTool
The full path to your build script (.bat or .sh) that contains the commands to build your source code. All references to the source code path in the build command must be replaced by the AWS IoT Device Tester variable `{ testData.sourcePath }`.

flashTool
Full path to your flash script (.sh or .bat) that contains the flash commands for your device. All references to the source code path in the flash command must be replaced by the AWS IoT Device Tester variable `{ testData.sourcePath }`.

clientWifiConfig
Client Wi-Fi configuration. The Wi-Fi library tests require an MCU board to connect to two access points. This attribute configures the Wi-Fi settings for the first access point. The client Wi-Fi settings are configured in `$AFR_HOME/tests/common/include/aws_clientcredential.h`. The following macros are set using the values found in `aws_clientcredential.h`. Some of the Wi-Fi test cases expect the access point to have some security and not to be open.

- `wifi_ssid`: The Wi-Fi SSID as a C string.
- `wifi_password`: The Wi-Fi password as a C string.
- `wifiSecurityType`: The type of Wi-Fi security used.

testWifiConfig
Test Wi-Fi configuration. The Wi-Fi library tests require a board to connect to two access points. This attribute configures the second access point. The test Wi-Fi settings are configured in `$AFR_HOME/tests/common/wifi/aws_test_wifi.c`. The following macros are set using the values found in `aws_test_wifi.c`. Some of the Wi-Fi test cases expect the access point to have some security and not to be open.

Note
If your board does not support Wi-Fi, you must still include the `clientWifiConfig` and `testWifiConfig` section in your `device.json` file, but you can omit values for these attributes.

- `testwifiWIFI_SSID`: The Wi-Fi SSID as a C string.
- `testwifiWIFI_PASSWORD`: The Wi-Fi password as a C string.
- `testwifiWIFI_SECURITY`: The type of Wi-Fi security used. One of the following values:
  - `eWiFiSecurityOpen`
  - `eWiFiSecurityWEP`
  - `eWiFiSecurityWPA`
  - `eWiFiSecurityWPA2`

otaConfiguration
The OTA configuration.
otaFirmwareFilePath

The full path to the OTA image created after the build.

deviceFirmwareFileName

The name of the OTA firmware file to be downloaded to the board.

awsSignerPlatform

The signing algorithm used by AWS Code Signer while creating the OTA update job. Currently, the possible values for this field are AmazonFreeRTOS-TI-CC3220SF and AmazonFreeRTOS-Default.

awsSignerCertificateArn

The Amazon Resource Name (ARN) for the trusted certificate uploaded to AWS Certificate Manager (ACM). For more information about creating a trusted certificate, see Creating a Code Signing Certificate.

awsUntrustedSignerCertificateArn

The ARN for the code-signing certificate uploaded to ACM.

compileCodesignerCertificate

Set to true if the code-signer signature verification certificate is not provisioned or flashed, so it must be compiled into the project. AWS IoT Device Tester fetches the trusted certificate from ACM and compiles it into aws_codesigner_certificate.h.

AWS IoT Device Tester Variables

The commands to build your code and flash the device might require connectivity or other information about your devices to run successfully. AWS IoT Device Tester allows you to reference device information in flash and build commands using JsonPath. By using simple JsonPath expressions, you can fetch the required information as specified in your device.json file.

AWS IoT Device Tester Variables and Concurrent Testing

To enable parallel builds of the source code for different test groups, AWS IoT Device Tester copies the source code to a results folder inside the AWS IoT Device Tester extracted folder. The source code path in your build or flash command must be referenced using the testdata.sourcePath variable. AWS IoT Device Tester replaces this variable with a temporary path of the copied source code.

File Paths and the Windows Operating System

If you are running AWS IoT Device Tester on Windows, use forward slashes (/) in your file paths in AWS IoT Device Tester config files. For example, sourcePath in userdata.json should be represented as c:/<dir1>/<dir2>.

Running the Amazon FreeRTOS Qualification Suite

You use the AWS IoT Device Tester executable to interact with AWS IoT Device Tester. The following command line shows you how to run the qualification tests for a device pool (set of identical devices).

devicetester_[linux | mac | win_x86-64] run-suite --suite-id <suite-id> --pool-id <your-device-pool> --userdata <userdata.json>

The userdata.json file should be located in the <devicetester_extract_location> / devicetester_afreertos_[win|mac|linux]/configs/ directory.
Note
If you are running AWS IoT Device Tester on Windows, specify the path to the userdata.json by using forward slashes (/).

Use the following command to run all test groups in a specified suite:

devicetester_[linux | mac | win_x86-64] run-suite --suite-id AFQ_1 --pool-id <pool-id>

Use the following command to run a specific test group:

devicetester_[linux | mac | win_x86-64] run-suite --suite-id AFQ_1 --group-id <group-id> --pool-id <pool-id>

suite-id and pool-id are optional if you are running a single test suite on a single device pool (that is, you have only one device pool defined in your device.json file).

AWS IoT Device Tester command line options

suite-id
Optional. Specifies the test suite to run.

pool-id
Specifies the device pool to test. If you only have one device pool, you can omit this option.

AWS IoT Device Tester Commands

The AWS IoT Device Tester command supports the following operations:

help
Lists information about the specified command.

list-groups
Lists the groups in a given suite.

list-suites
Lists the available suites.

run-suite
Runs a suite of tests on a pool of devices.

Results and Logs

This section describes how to view and interpret test results and logs.

Viewing Results

After AWS IoT Device Tester executes the qualification test suite, it generates two reports for each run of the qualification test suite: awsiotdevicetester_report.xml and AFQ_Report.xml. These reports can be found in <devicetester-extract-location>/results/<execution-id>/.

awsiotdevicetester_report.xml is the qualification test report that you submit to AWS for listing your device in the AWS Partner Device Catalog. The report contains the following elements:
• The AWS IoT Device Tester version.
• The Amazon FreeRTOS version that was tested.
• The SKU and the device name specified in the device.json file.
• The features of the device specified in the device.json file.
• The aggregate summary of test case results.
• A breakdown of test case results by libraries that were tested based on the device features (for example, FullWiFi, FullMQTT, and so on).

The AFQ_report.xml is a report in standard junit.xml format, which you can integrate into your exiting CI/CD platforms like Jenkins, Bamboo, and so on. The report contains the following elements:

• An aggregate summary of test case results.
• A breakdown of test case results by libraries that were tested based on the device features (for example, FullWiFi, FullMQTT, and so on).

Interpreting AWS IoT Device Tester Results

The report section in awsiotdevicetester_report.xml or AFQ_report.xml lists the tests that were run and the results of the tests.

The first XML tag <testsuites> contains the overall summary of the test execution. For example:

```
<testsuites name="AFQ results" time="5633" tests="184" failures="0" errors="0" disabled="0">
```

**Attributes used in the <testsuites> tag**

- **name**
  - The name of the test suite.
- **time**
  - The time (in seconds) it took to run the qualification suite.
- **tests**
  - The number of test cases executed.
- **failures**
  - The number of test cases that were run, but did not pass.
- **errors**
  - The number of test cases that AWS IoT Device Tester couldn't execute.
- **disabled**
  - This attribute is not used and can be ignored.

If there are no test case failures or errors, your device meets the technical requirements to run Amazon FreeRTOS and can interoperate with AWS IoT services. If you choose to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the case of test case failures or errors, you can identify the test case that failed by reviewing the <testsuites> XML tags. The <testsuite> XML tags inside the <testsuites> tag shows the test case result summary for a test group.
<testsuite name="FullMQTT" package="" tests="16" failures="0" time="76" disabled="0" errors="0" skipped="0">

The format is similar to the <testsuites> tag, but with an additional attribute called skipped that is not used and can be ignored. Inside each <testsuite> XML tag, there are <testcase> tags for each of the test cases that were executed for a test group. For example:

<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase" attempts="1"></testcase>

**Attributes used in the <testcase> tag**

- **name**: The name of the test case.
- **attempts**: The number of times AWS IoT Device Tester executed the test case.

When a test case fails or an error occurs, <failure> or <error> tags are added to the <testcase> tag with additional information for troubleshooting. For example:

<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase" attempts="1">
  <failure type="Failure">Reason for the test case failure</failure>
  <error>Reason for the test case execution error</error>
</testcase>

**Viewing Logs**

You can find logs that AWS IoT Device Tester generates from test execution in `<devicetester-extract-location>/results/<execution-id>/logs`. Two sets of logs are generated:

- **test_manager.log**: Contains logs generated from the Test Manager component of AWS IoT Device Tester. For example, logs related configuration, test sequencing, and report generation are here.
- **<test_group_name>.log** (for example, Full_MQTT.log)

  The logs of the test group, including logs from the device under test.

**Test for Requalifications**

As new versions of AWS IoT Device Tester qualification tests are released, as you update your board-specific packages or device drivers, you can use AWS IoT Device Tester to test your microcontroller boards. For subsequent qualifications, make sure that you have the latest versions of Amazon FreeRTOS and AWS IoT Device Tester and run the qualification tests again.:

**Troubleshooting**

We recommend the following workflow for troubleshooting testing an Amazon FreeRTOS device:

1. Read the console output.
3. Look in the logs files located under /results/<uuid>/logs.
4. Investigate one of the following problem areas:
   - Device configuration
   - Device interface
   - Device tooling
   - Amazon FreeRTOS source code

Troubleshooting Device Configuration

When you use AWS IoT Device Tester, you must get the correct configuration files in place before you execute the binary. If you are getting parsing and configuration errors, your first step should be to locate and use a configuration template appropriate for your environment.

If you are still having issues, see the debugging following process.

Where Do I Look?

Start by looking in the results.xml file in the /results/<uuid> directory. This file contains all of the test cases that were run and error snippets for each failure. To get all of the execution logs, look under /results/<uuid>/<test-case-id>.log for each test group.

Parsing Errors

Occasionally, a typo in a JSON configuration can lead to parsing errors. Most of the time, the issue is a result of omitting a bracket, comma, or quote from your JSON file. AWS IoT Device Tester performs JSON validation and prints debugging information. It prints the line where the error occurred, the line number, and the column number of the syntax error. This information should be enough to help you fix the error, but if you are still having issues locating the error, you can perform validation manually in your IDE, a text editor such as Atom or Sublime, or through an online tool like JSONLint.

Required Parameter Missing Error

Because new features are being added to AWS IoT Device Tester, changes to the configuration files might be introduced. Using an old configuration file might break your configuration. If this happens, the <test_case_id>.log file under /results/<uuid>/logs explicitly lists all missing parameters. AWS IoT Device Tester validates your JSON configuration file schemas to ensure that the latest supported version has been used.

Could Not Start Test Error

You might encounter errors that point to failures during test start. Since there are several possible causes for this, please make sure to check the following areas for correctness:

- Make sure that the pool name you've included in your execution command actually exists. This is referenced directly from your device.json file.
- Ensure that the device(s) in your pool have correct configuration parameters.

Device Interface and Port

This section contains information about the device interfaces AWS IoT Device Tester uses to connect to your devices.
Supported Platforms

AWS IoT Device Tester supports Linux, macOS, and Windows. All three platforms have different naming schemes for serial devices that are attached to them:

- Linux: /dev/tty*
- macOS: /dev/tty.*
- Windows: COM*

To check your device ID:

- For Linux/macOS, open a terminal and run `ls /dev/tty*`.
- For Windows, open Device Manager and expand the serial devices group.

Device Interfaces

Each embedded device is different, which means that they can have one or more serial ports. It is common for devices to have two ports when connected to a machine, one being a data port for flashing the device and the other to read output. It is crucial to set the correct port in your `device.json` file. Otherwise, flashing or reading output might fail.

In the case of multiple ports, make sure to use the data port of the device in your `device.json` file. For example, if you plug in an Espressif WRover device and the two ports assigned to it are /dev/ttyUSB0 and /dev/ttyUSB1, use /dev/ttyUSB1 in your `device.json` file.

For Windows, follow the same logic.

Reading Device Data

AWS IoT Device Tester uses individual device build and flash tooling to specify port configuration. If you are testing your device and don't get output, try the following default settings:

- **Baud rate**: 115200
- **Data Bits**: 8
- **Parity**: None
- **Stop Bits**: 1
- **Flow Control**: None

These settings are handled by AWS IoT Device Tester without any configuration on your end. However, you can use the same method to manually read device output. On Linux or macOS, you can do this with the `screen` command. On Windows, you can use a program such as TeraTerm.

- **Screen**: `screen /dev/cu.usbserial 115200`
- **TeraTerm**: Use the above-provided settings to set the fields explicitly in the GUI.

Development Toolchain Problems

This section discusses problems that can occur with your toolchain.

Code Composer Studio on Ubuntu

For TI devices, we recommend that you download and install Code Composer Studio 7.3. The newer versions of Ubuntu (17.10 and 18.04) have a version of the glibc package that is not compatible with Code Composer Studio 7.x versions. We recommended that you install Code Composer Studio version 8.2 or later.
Symptoms of incompatibility might include:

- Amazon FreeRTOS failing to build or flash to your device.
- The Code Composer Studio installer might freeze.
- No log output is displayed in the console during the build or flash process.
- Build command attempting to launch in GUI mode even when invoked as headless.

Amazon FreeRTOS Source Code

The following sections discuss troubleshooting problems with the Amazon FreeRTOS source code.

Code Errata

Every Amazon FreeRTOS release is bundled with a document, located under the /amazon-freertos/tests directory, that contains all of the errata information for that release. We recommend that you read through this document before you run any tests. The errata document contains an entry for any devices that currently fail tests due to reasons like:

- The hardware doesn't support a specific feature.
- The hardware supports the feature, but Amazon FreeRTOS doesn't support it on the device yet.
- The hardware supports the feature, but the underlying software stack doesn't support the hardware (non-AFR).

If the errata does not contain information specific to your device, continue the debugging process as outlined in the next section.

Debugging Amazon FreeRTOS

When a source code error occurs, AWS IoT Device Tester will write debug output to the <test-group-id>.log file in the /results/<uuid>/logs directory. Search the file for any instances of errors. The error will point to a location in the Amazon FreeRTOS source code. You can then use the line number and file path information in that log to reference the piece of source code that resulted in the error.

Logging

AWS IoT Device Tester logs are placed in a single location. From the root AWS IoT Device Tester directory, the available files are:

- ./results/<uuid>
- AFQ_Report.xml
- awsiotdevicetester_report.xml
- /logs/<test_group_id>.log

The most important logs to look at will be <test_group_id>.log and results.xml. The latter will contain information about which tests failed with a specific error message. You can then use the former to dig further into the problem in order to get better context.

Console Errors

When AWS IoT Device Tester is run, failures are reported to console with brief messages. Look in <test_group_id>.log to learn more about the error.
Log Errors

The <test-group-id>.log file is located in the /results/<uuid> directory. Each test execution has a unique test ID that is used to create the <uuid> directory. Individual test group logs are under the <uuid> directory. Use the AWS IoT console to look up the test group that failed and then open the log file for that group in the /results/<uuid> directory. The information in this file includes the full build and flash command output, as well as test execution output and more verbose AWS IoT Device Tester console output.

Path Variables

AWS IoT Device Tester defines the following path variables that can be used in command lines and configuration files:

{{testData.sourcePath}}
A variable that expands to the source code path.

{{device.connectivity.serialPort}}
A variable that expands to the serial port.

{{device.identifiers[?(@.name == 'serialNo')].value}}
A variable that expands to the serial number of your device.

The following is an example userdata.json file:

```json
{
    "sourcePath": "<path/to/amazon-freertos>",
    "buildTool": {
        "name": "<TOOL_NAME>",
        "version": "<TOOL_VERSION>",
        "command": [
            "<path/to/build>.sh {{testData.sourcePath}}"
        ]
    },
    "flashTool": {
        "name": "<TOOL_NAME>",
        "version": "<TOOL_VERSION>",
        "command": [
            "<path/to/flash>.sh {{device.connectivity.serialPort}} {{testData.sourcePath}}"
        ]
    },
    "clientWifiConfig": {
        "wifiSSID": "<SSID1>",
        "wifiPassword": "<PASSWORD>",
        "wifiSecurityType": "eWiFiSecurityWPA2"
    },
    "testWifiConfig": {
        "wifiSSID": "<SSID2>",
        "wifiPassword": "<PASSWORD>",
        "wifiSecurityType": "eWiFiSecurityWPA2"
    },
    "otaConfiguration": {
        "otaFirmwareFilePath": "{{testData.sourcePath}}/ota-image/from/root/of/afrsourcecode",
        "deviceFirmwareFileName": "<deviceFirmwareFileName>",
        "awsSignerPlatform": "AmazonFreeRTOS-Default",
        "awsSignerCertificateArn": "arn:aws:acm::<region>::<account-id>::certificate::<certificate-Id>"
    }
}
```
The following is an example device.json file:

```json
[
  {
    "id": "<POOL_NAME>",
    "sku": "<armsku>",
    "features": [
      {
        "name": "WIFI",
        "value": "<Yes>"
      },
      {
        "name": "OTA",
        "value": "<Yes>"
      },
      {
        "name": "TCP/IP",
        "value": "Offloaded"
      },
      {
        "name": "TLS",
        "value": "On-chip"
      }
    ],
    "devices": [
      {
        "id": "<DEVICE_NAME>",
        "connectivity": {
          "protocol": "uart",
          "serialPort": "/dev/tty<PORT>" OR "/dev/tty.<PORT>"
        },
        "identifiers": [
          {
            "name": "serialNo",
            "value": "<ABCDEZAGHJI>"
          }
        ]
      }
    ]
  }
]
```

On the Windows platform, the userdata and device configuration files are formatted in the same manner. Pay close attention to the direction of the path-separator slashes. We recommend using the forward slash (/) because newer versions of Windows support it. If you are using Windows 7 or earlier, use the back slash (\).

### Permissions Policy Template

The following is a policy template that grants the permissions required for AWS IoT Device Tester:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Principal": {
        "Service": "iot:四方设备测试器"
      }
    }
  ]
}
```
"Statement": [
{
"Sid": "VisualEditor0",
"Effect": "Allow",
"Action": [
"iam:CreatePolicy",
"iam:DetachRolePolicy",
"iam:DeleteRolePolicy",
"s3:CreateBucket",
"iam:DeletePolicy",
"iam:CreateRole",
"iam:DeleteRole",
"iam:AttachRolePolicy",
"s3:DeleteBucket",
"s3:PutBucketVersioning"
],
"Resource": [
"arn:aws:s3:::idt*",
"arn:aws:s3:::afr-ota*",
"arn:aws:iam::*:policy/idt*",
"arn:aws:iam::*:role/idt*"
]
},
{
"Sid": "VisualEditor1",
"Effect": "Allow",
"Action": [
"iot:DeleteCertificate",
"iot:AttachPolicy",
"iot:DetachPolicy",
"s3:DeleteObjectVersion",
"iot:DeleteOTAUpdate",
"s3:PutObject",
"s3:GetObject",
"iam:PassRole",
"iot:DeleteStream",
"iot:DeletePolicy",
"iot:UpdateCertificate",
"iot:GetOTAUpdate",
"s3:DeleteObject",
"iot:DescribeJobExecution",
"s3:GetObjectVersion"
],
"Resource": [
"arn:aws:iot:*:*:thinggroup/idt*",
"arn:aws:iot:*:*:policy/idt*",
"arn:aws:iot:*:*:otaupdate/idt*",
"arn:aws:iot:*:*:thing/idt*",
"arn:aws:iot:*:*:cert/*",
"arn:aws:iot:*:*:job/*",
"arn:aws:iot:*:*:stream/*",
"arn:aws:iam::*:role/idt*",
"arn:aws:s3:::afr-ota/*",
"arn:aws:s3:::idt/*",
"arn:aws:iam::*:role/idt*"
]
},
{
"Sid": "VisualEditor2",
"Effect": "Allow",
"Action": [
"iot:DetachThingPrincipal",
"iot:AttachThingPrincipal",
"s3:ListBucketVersions",
"iot:CreatePolicy",
"iam:ListRoles",
"iam:CreatePolicy",
"iam:DeletePolicy",
"iam:CreateRole",
"iam:DeleteRole",
"iam:AttachRolePolicy",
"s3:DeleteBucket",
"s3:PutBucketVersioning"
],
"Resource": [
"arn:aws:s3:::idt*",
"arn:aws:s3:::afr-ota*",
"arn:aws:iam::*:policy/idt*",
"arn:aws:iam::*:role/idt*"
]
},
]}
"freertos:ListHardwarePlatforms",
"signer:DescribeSigningJob",
"s3:ListBucket",
"signer:*",
"iot:DescribeEndpoint",
"iot:CreateStream",
"signer:StartSigningJob",
"s3:ListAllMyBuckets",
"signer:ListSigningJobs",
"acm:GetCertificate",
"acm:ListCertificates",
"acm:ImportCertificate",
"freertos:DescribeHardwarePlatform",
"iot:CreateKeysAndCertificate",
"iot:CreateCertificateFromCsr",
"s3:GetBucketLocation",
"iot:GetRegistrationCode",
"iot:RegisterCACertificate",
"iot:RegisterCertificate",
"iot:UpdateCACertificate",
"iot:DeleteCACertificate",
"iot:DeleteCertificate",
"iot:UpdateCertificate"
],
"Resource": "*
",
{
"Sid": "VisualEditor3",
"Effect": "Allow",
"Action": [
"s3:PutObject",
"s3:GetObject"
],
"Resource": [
"arn:aws:s3:::afr*/*",
"arn:aws:s3:::idt*/*
]
},
{
"Sid": "VisualEditor4",
"Effect": "Allow",
"Action": [
"iot:CreateOTAUpdate",
"iot:CreateThing",
"iot:DeleteThing"
],
"Resource": "*
"
}