Amazon FreeRTOS

User Guide

Amazon FreeRTOS: User Guide

Table of Contents

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 configuration wizard 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 Greengrass cores.
- Manage Wi-Fi connections.

Amazon FreeRTOS Configuration Wizard

The Amazon FreeRTOS configuration wizard 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 configuration wizard 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 configuration wizard

1. Browse to the AWS IoT console.
2. From the navigation pane choose **Software**.
3. Under **Amazon FreeRTOS Device Software** choose **Configure Download**.

Over-the-Air Updates (Beta)

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:

- Digitially 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.

For more information about OTA updates, see:

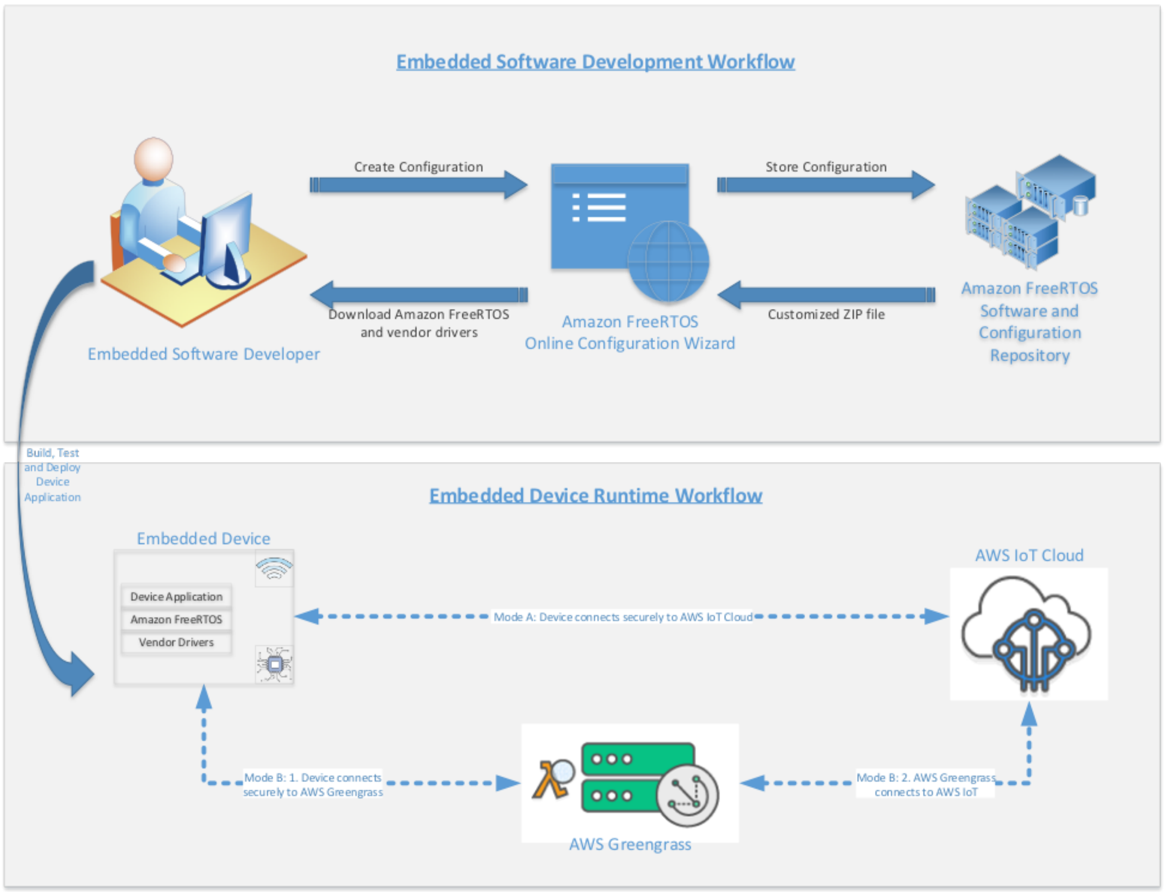
- Amazon FreeRTOS Over-the-Air Updates (p. 67)
- OTA Demo Application (p. 32)

Amazon FreeRTOS Qualification Program

The Amazon FreeRTOS Qualification Program (Amazon FQP) is for microcontroller vendors who want to qualify their microcontroller-based hardware on Amazon FreeRTOS. The goal of Amazon FQP is to ensure that developers can use Amazon FreeRTOS on their choice of microcontroller-based hardware. In order to deliver a consistent experience for developers, the Amazon FQP outlines a set of security, functionality, and performance requirements that all microcontrollers (and the associated hardware abstraction layers and drivers) must meet.

Development Workflow

You start development by configuring and downloading a package from the Amazon FreeRTOS configuration wizard 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 Greengrass as part of a complete IoT solution. The following diagram shows the development workflow and the subsequent connectivity from Amazon FreeRTOS-based devices.


Embedded Software Development Workflow
Create Configuration
Store Configuration
Download Amazon FreeRTOS and vendor drivers
Customized ZIP file
Embedded Software Developer
Amazon FreeRTOS Online Configuration Wizard
Amazon FreeRTOS Software and Configuration Repository
Build, Test and Deploy Device Application
Embedded Device Runtime Workflow
Embedded Device
Device Application
Amazon FreeRTOS
Vendor Drivers
AWS IoT Cloud
Mode A: Device connects securely to AWS IoT Cloud
Mode B: 1. Device connects securely to AWS Greengrass
Mode B: 2. AWS Greengrass connects to AWS IoT
AWS Greengrass

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

Before you begin, 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. IAM users must be granted access to AWS IoT and Amazon FreeRTOS. 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 left navigation pane, choose **Users**.
2. Type your user name in the search text box, and then choose your user name from the list.
3. Choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. In the search box, type **`AmazonFreeRTOSFullAccess`**, select 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 left navigation pane, choose **Users**.
2. Type your user name in the search text box, and then choose your user name from the list.
3. Choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. In the search box, type **`AWSIoTFullAccess`**, select 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:

- STMicroelectronicsSTM32L4 Discovery kit IoT node
- Texas Instruments CC3220SF-LAUNCHXL
- NXP LPC54018 IoT Module
- Microchip Curiosity PIC32MZEF bundle
- Microsoft Windows 7 or later, with at least a dual core and a hard-wired Ethernet connection

Registering Your MCU Board with AWS IoT

You must register your MCU board so it can communicate with AWS IoT. Registering your device involves creating an IoT thing, a private key, an X.509 certificate, and an AWS IoT policy. An IoT thing allows you to manage your devices in AWS IoT. The private key and certificate allow your device to authenticate with AWS IoT. 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 region your AWS account is in, open a command prompt window and type the following command:

   ```
   aws iot describe-endpoint
   ```

   The output should look like this:

   ```
   {
       "endpointAddress": "xxxxxxxxxxxxxx.iot.us-west-2.amazonaws.com"
   }
   ```

   In this example, the region is `us-west-2`.
2. Browse to the AWS IoT console.
3. In the left navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**.
4. Type 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 .

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": "iot:Connect",
        "Resource":"arn:aws:iot:<aws-region>:<aws-account-id>:*"
```

```
 },
    {
        "Effect": "Allow",
        "Action": "iot:Publish",
        "Resource": "arn:aws:iot:<aws-region>:<aws-account-id>:*"
    },
    {
         "Effect": "Allow",
         "Action": "iot:Subscribe",
         "Resource": "arn:aws:iot:<aws-region>>:<aws-account-id>:*"
    },
    {
         "Effect": "Allow",
         "Action": "iot:Receive",
         "Resource": "arn:aws:iot:<aws-region>:<aws-account-id>:*"
    }
    ]
}
```

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 left 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**.
5. On the **Add your device to the thing registry** page, type a name for your thing, and then choose **Next**.
6. On the **Add a certificate for your thing** page, under **One-click certificate creation**, choose **Create certificate**.
7. Download your private key and certificate by choosing the **Download** links for each. Make a note of the certificate ID. You need it later to attach a policy to your certificate.
8. Choose **Activate** to activate your certificate. Certificates must be activated prior to use.
9. Choose **Done**.
10. Choose **Attach a policy** to attach a policy to your certificate that grants your device access to AWS IoT operations.
11. Choose the policy you just created and choose **Register thing**.

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. Before you begin, install one of the two IDEs:

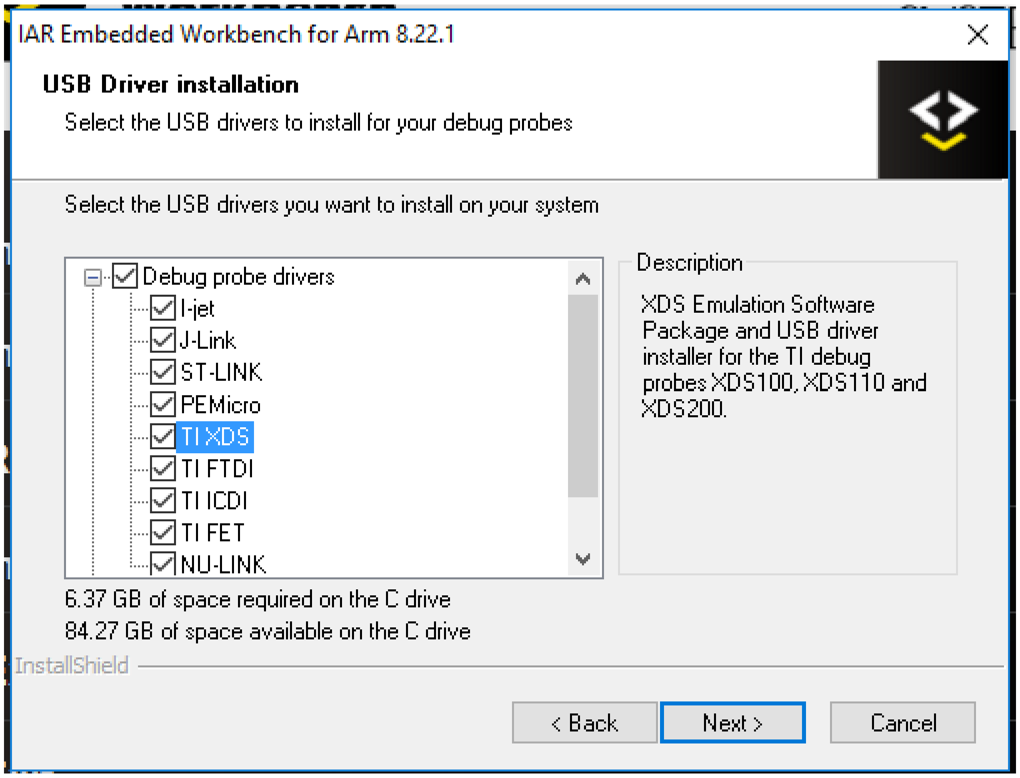
Option 1: Install Texas Instruments Code Composer Studio

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.

Option 2: Install IAR Embedded Workbench for ARM

1. Browse to IAR Embedded Workbench for ARM.
2. Download and run the Windows installer. Make sure that **TI XDS** is selected as one of the USB Debug probe drivers:


IAR Embedded Workbench for Arm 8.22.1

USB Driver installation

Select the USB drivers to install for your debug probes

Select the USB drivers you want to install on your system

Description

XDS Emulation Software Package and USB driver installer for the TI debug probes XDS100, XDS110 and XDS200.

6.37 GB of space required on the C drive
84.27 GB of space available on the C drive

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

Download and Build Amazon FreeRTOS

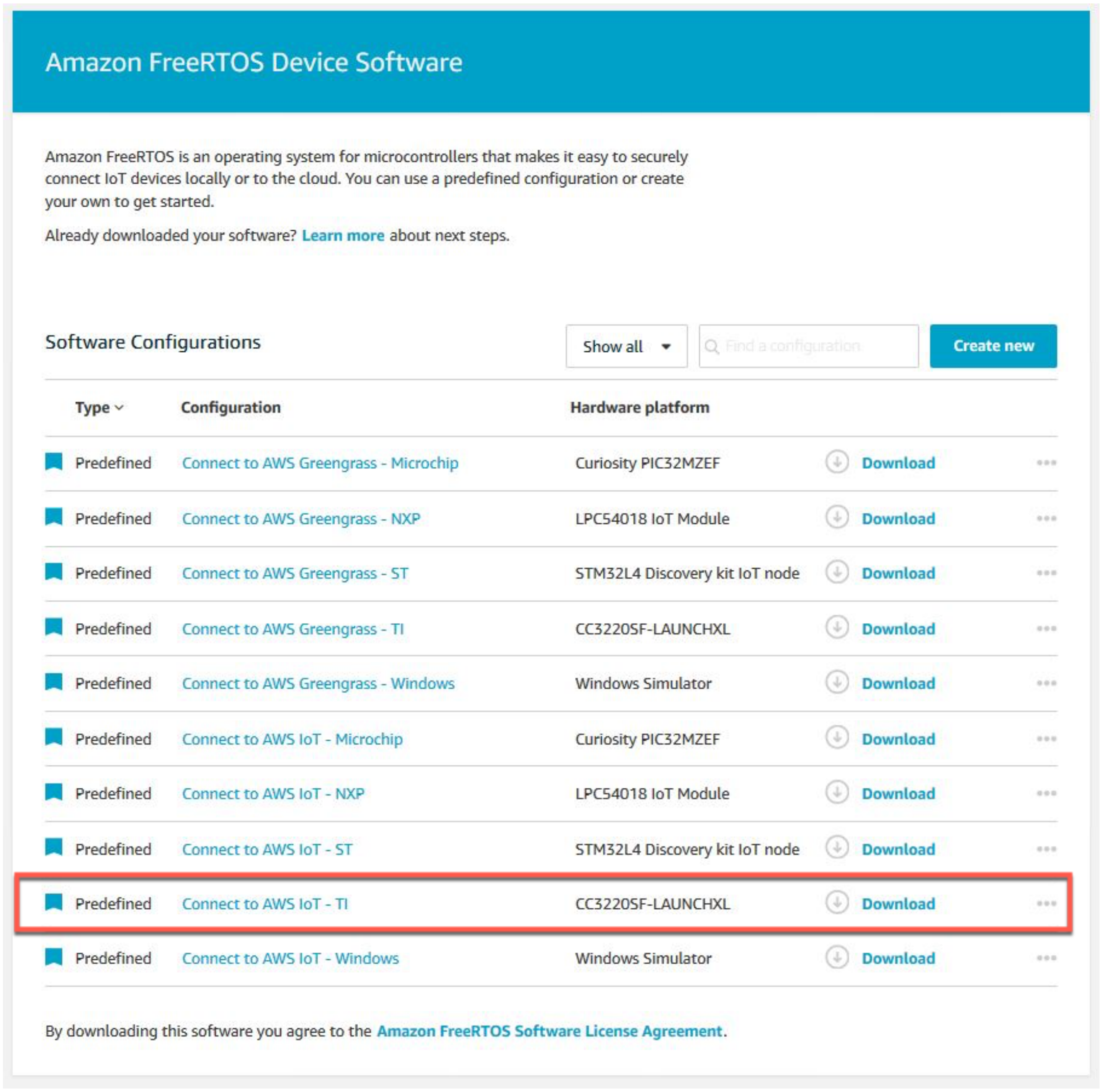
After your environment is set up, you can download Amazon FreeRTOS and run the sample code.

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- TI**, and then:

   Option 1 (if you are using CCS): choose **Download**.

   Option 2 (if you are using IAR): choose **Connect to AWS IoT-TI**. Under **Hardware platform**, choose **Edit**. Under **Integrated Development Environment (IDE)**, choose **IAR Embedded Workbench** from the drop-down list. Make sure the compiler is set to IAR. Then choose **Create and Download**.



Amazon FreeRTOS Device Software

Amazon FreeRTOS is an operating system for microcontrollers that makes it easy to securely connect IoT devices locally or to the cloud. You can use a predefined configuration or create your own to get started.

Already downloaded your software? Learn more about next steps.

Software Configurations

Type	Configuration	Hardware platform		
Predefined	Connect to AWS Greengrass - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS Greengrass - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS Greengrass - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS Greengrass - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS Greengrass - Windows	Windows Simulator	Download	•••
Predefined	Connect to AWS IoT - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS IoT - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS IoT - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS IoT - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS IoT - Windows	Windows Simulator	Download	•••

By downloading this software you agree to the Amazon FreeRTOS Software License Agreement.

5. Unzip the downloaded file to a folder, and make a note of the folder path. In this tutorial, this folder is referred to as `BASE_FOLDER`.

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 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.

Import the Amazon FreeRTOS Sample Code into TI Code Composer

(Follow these steps if you are using 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 the **Select search-directory** text box, type `<BASE_FOLDER>\AmazonFreeRTOS\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. From the **Project** menu, choose **Build Project** to make sure it compiles without errors or warnings.

Import the Amazon FreeRTOS Sample Code into IAR Embedded Workbench

(Follow these steps if you are using IAR Embedded Workbench.)

1. Open IAR Embedded Workbench and choose **File > Open Workspace**.
2. Navigate to `<BASE_FOLDER>\AmazonFreeRTOS\demos\ti\cc3220_launchpad\iar` and choose **aws_demos.eww** then choose **OK**.
3. Right-click on the project name (`aws_demos`) then choose **Make**.

Configure Your Project

In the **Project Explorer** window, open `aws_demos\application_code\common_demos\include\aws_clientcredential.h`.

To configure your project you need to know your AWS IoT endpoint.

To find your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the left navigation pane, choose **Settings**.
3. Copy your AWS IoT endpoint from the **Endpoint** text box. It should look like `<1234567890123>.iot.<us-east-1>.amazonaws.com`.
4. Open `aws_demos/application_code/common_demos/include/aws_clientcredential.h` and set `clientcredentialMQTT_BROKER_ENDPOINT` to your AWS IoT endpoint. Save your changes.

You also need to know your Wi-FI network SSID, password, and security type, and the name of the AWS IoT thing that represents your device. Valid security types are:

- `eWiFiSecurityOpen`: Open, no security.
- `eWiFiSecurityWEP`: WEP security.
- `eWiFiSecurityWPA`: WPA security.
- `eWiFiSecurityWPA2`: WPA2 security.

In the **Project Explorer** window, open `aws_demos\application_code\common_demos\include\aws_clientcredential.h`.

Specify values for the following `#define` constants:

- `clientcredentialMQTT_BROKER_ENDPOINT`: Your AWS IoT endpoint.
- `clientcredentialIOT_THING_NAME`: The AWS IoT thing for your board.
- `clientcredentialWIFI_SSID`: The SSID for your Wi-Fi network.

- `clientcredentialWIFI_PASSWORD`: The password for your Wi-Fi network.
- `clientcredentialWIFI_SECURITY`: The security type for your Wi-Fi network.

Make sure to save your changes.

Configure Your AWS IoT Credentials

The certificate and private key must be hard-coded into the Amazon FreeRTOS demo code. Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project.

To format your certificate and private key

1. In a browser window, open `<BASE_FOLDER>\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the `<ID>-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the `<ID>-private.pem.key` 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 stub file in the directory.

Run the FreeRTOS Samples

1. Make sure the SOP jumper on your Texas Instruments CC3220SF-LAUNCHXL is in position 0.
2. Use a USB cable to connect your Texas Instruments CC3220SF-LAUNCHXL to your computer.
3. Rebuild your project.
4. Sign in to the AWS IoT console.
5. In the left navigation pane, choose **Test** to open the MQTT client.
6. In the **Subscription topic** text box, type **`freertos/demos/echo`**, and then choose **Subscribe to topic**.
7. From the **Run** menu in TI Code Composer, choose **Debug** to start debugging.
8. When the debugger stops at the breakpoint in `main()`, go to the **Run** menu, and then 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 received 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 left navigation pane, choose **aws_demos**.
4. Choose the **Target** tab in the main window.
5. Scroll down to the **Connection Options** section and select the **Reset the target on a connect** check box.
6. Choose **Apply**, and then choose **Close** to close the **Debug Configurations** dialog box.

If your device hangs after calling `sl_Start()`, you might need to update the service pack running on the network processor on the TI chip.

To update the 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**, and then choose **Start Image Creator**.
3. Choose **New Project**.
4. On the **Start new project** page, type a name for your project. Set **Device Type** to **CC3220SF**. Set **Device Mode** to **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_`*`VERSION`*`\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.

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.

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.
3. Browse to the System Workbench for STM32 installer to download and install System Workbench.

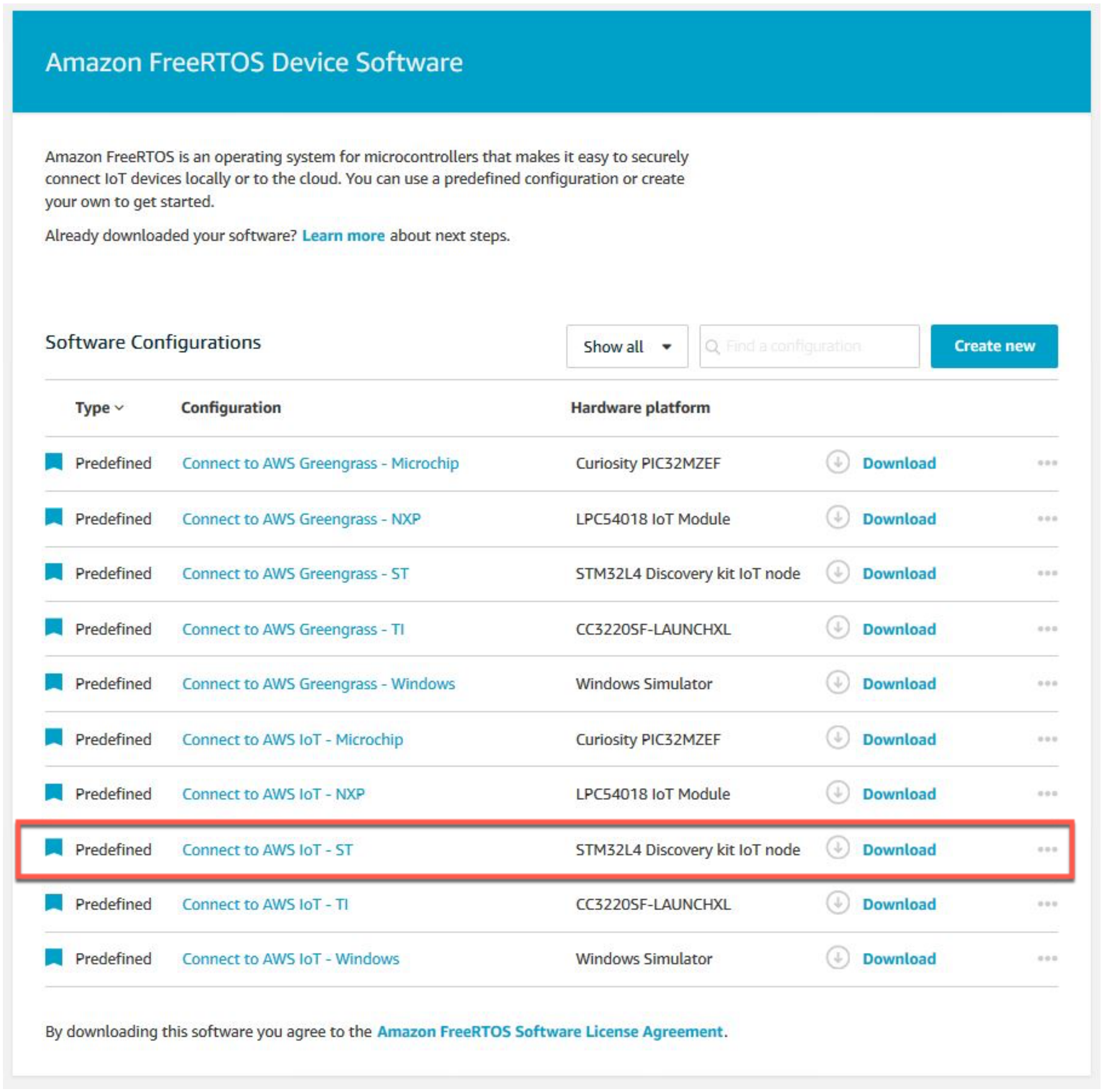
If you experience issues during installation, see the FAQs on the System Workbench website.

Download and Build Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS and run the sample code.

Download Amazon FreeRTOS

1. In the AWS IoT console, browse to the Amazon FreeRTOS page.
2. In the left 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- ST**, and then choose **Download**.



Amazon FreeRTOS Device Software

Amazon FreeRTOS is an operating system for microcontrollers that makes it easy to securely connect IoT devices locally or to the cloud. You can use a predefined configuration or create your own to get started.

Already downloaded your software? Learn more about next steps.

Software Configurations

Type	Configuration	Hardware platform		
Predefined	Connect to AWS Greengrass - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS Greengrass - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS Greengrass - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS Greengrass - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS Greengrass - Windows	Windows Simulator	Download	•••
Predefined	Connect to AWS IoT - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS IoT - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS IoT - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS IoT - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS IoT - Windows	Windows Simulator	Download	•••

By downloading this software you agree to the Amazon FreeRTOS Software License Agreement.

6. Unzip the downloaded file to a folder, and make a note of the folder path. In this tutorial, this folder is referred to as `BASE_FOLDER`.

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 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.

Import the Amazon FreeRTOS Sample Code into the STM32 System Workbench

1. Open the STM32 System Workbench and type 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 the **Select Root Directory** text box, type *<BASE_FOLDER>*\AmazonFreeRTOS\demos\st\stm32l475_discovery\ac6.
4. The project aws_demos should be found and selected by default.
5. Choose **Finish** to import the project into STM32 System Workbench.
6. From the **Project** menu, choose **Build All** and make sure it compiles without any errors or warnings.

Configure Your Project

To configure your project, you need to know your AWS IoT endpoint.

To find your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the left navigation pane, choose **Settings**.
3. Copy your AWS IoT endpoint from the **Endpoint** text box. It should look like *<1234567890123>*.iot.*<us-east-1>*.amazonaws.com.
4. Open aws_demos/application_code/common_demos/include/aws_clientcredential.h and set clientcredentialMQTT_BROKER_ENDPOINT to your AWS IoT endpoint. Save your changes.

You also need to know your Wi-FI network SSID, password, and security type and the name of the AWS IoT thing that represents your device. Valid security types are:

- eWiFiSecurityOpen: Open, no security.
- eWiFiSecurityWEP: WEP security.
- eWiFiSecurityWPA: WPA security.
- eWiFiSecurityWPA2: WPA2 security.

In the **Project Explorer** window, open aws_demos\application_code\common_demos\include\aws_clientcredential.h.

Specify values for the following #define constants:

- clientcredentialMQTT_BROKER_ENDPOINT: Your AWS IoT endpoint.
- clientcredentialIOT_THING_NAME: The AWS IoT thing for your board.
- clientcredentialWIFI_SSID: The SSID for your Wi-Fi network.
- clientcredentialWIFI_PASSWORD: The password for your Wi-Fi network.
- clientcredentialWIFI_SECURITY: The security type for your Wi-Fi network.

Make sure to save your changes.

Configure Your AWS IoT Credentials

The certificate and private key must be hard-coded into the Amazon FreeRTOS demo code. Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project.

To format your certificate and private key

1. In a browser window, open *<BASE_FOLDER>*`\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the *<ID>*`-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the *<ID>*`-private.pem.key` 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 stub file in the directory.

Run the 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 left navigation pane, choose **Test** to open the MQTT client.
5. In the **Subscription topic** text box, type `freertos/demos/echo`, and then choose **Subscribe to topic**.
6. From the **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. In list box on the left, 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**.

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

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 the two IDEs:

To install IAR Embedded Workbench for Arm

1. Browse to Software for NXP Kits and choose **Download Software** to install IAR Embedded Workbench for Arm.

   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**.
4. Under **Select Development Board**, type `LPC54018-IoT-Module` in the **Search by Name** text box.
5. Under **Boards** choose **LPC54018-IoT-Module**.
6. On the right-hand side of the page, verify the hardware details and choose **Build MCUXepresso 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. In the bottom of the MCUXpresso IDE screen, choose the **Installed SDKs** tab.
9. Drag and drop the downloaded sdk archive file into the **Installed SDKs** window.

Note
If you experience issues during installation, see NXP Support or NXP Developer Resources.

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.

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 Build Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS and run the sample code.

Download Amazon FreeRTOS

1. Browse to the Amazon FreeRTOS page in the AWS IoT console.
2. In the left navigation pane, choose **Software**.
3. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.

4. Choose **Download FreeRTOS Software**.
5. If you are using IAR Workbench:
   - In the **Software Configurations**, find **Connect to AWS IoT- NXP** and choose **Download**.
6. If you are using MCUXpresso:
   a. In the **Software Configurations**, find **Connect to AWS IoT- NXP**. Select the **Connect to AWS IoT- NXP** text, 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**.
7. Unzip the downloaded file to a folder and make a note of the folder path. In this tutorial, this folder is referred to as `BASE_FOLDER`.

   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 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.

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, type `<BASE_FOLDER>\AmazonFreeRTOS\demos\nxp\lpc54018_iot_module\iar`, and choose **aws_demos.eww**.
3. In 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, type *<BASE_FOLDER>*\AmazonFreeRTOS\demos\nxp\lpc54018_iot_module\mcuxpresso and choose **Finish**
3. In the **Project** menu, choose **Build All**.

Configure Your Project

To configure your project, you need to know your AWS IoT endpoint.

To find your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the left navigation pane, choose **Settings**.
3. Copy your AWS IoT endpoint from the **Endpoint** text box. It should look like `<1234567890123>.iot.<us-east-1>.amazonaws.com`.
4. Open `aws_demos/application_code/common_demos/include/aws_clientcredential.h` and set `clientcredentialMQTT_BROKER_ENDPOINT` to your AWS IoT endpoint. Save your changes.

You also need to know your Wi-FI network SSID, password, and security type and the name of the AWS IoT thing that represents your device. Valid security types are:

- `eWiFiSecurityOpen`: Open, no security.
- `eWiFiSecurityWEP`: WEP security.
- `eWiFiSecurityWPA`: WPA security.
- `eWiFiSecurityWPA2`: WPA2 security.

In your IDE, open `aws_demos\application_code\common_demos\include\aws_clientcredential.h`.

Specify values for the following `#define` constants:

- `clientcredentialMQTT_BROKER_ENDPOINT`: Your AWS IoT endpoint.
- `clientcredentialIOT_THING_NAME`: The AWS IoT thing for your board.
- `clientcredentialWIFI_SSID`: The SSID for your Wi-Fi network.
- `clientcredentialWIFI_PASSWORD`: The password for your Wi-Fi network.
- `clientcredentialWIFI_SECURITY`: The security type for your Wi-Fi network.

Make sure to save your changes.

Configure Your AWS IoT Credentials

The certificate and private key must be hard-coded into the Amazon FreeRTOS demo code. Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project.

To format your certificate and private key

1. In a browser window, open *<BASE_FOLDER>*`\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the *<ID>*`-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the *<ID>*`-private.pem.key` 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 stub file in the directory.

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 left navigation pane, choose **Test** to open the MQTT client.
4. In the **Subscription topic** text box, type `freertos/demos/echo`, and then choose **Subscribe to topic**.
5. In your IDE, from the **Project** menu, choose **Build**.
6. Connect both 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`, go to the **Debug** menu and 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 be do this once.
8. If you are using MCUXpresso:

   a. If this is your first time debugging, select the `aws_demos` project and from the **Debug** toolbar, choose the blue debug button (Start debugging the project with the selected build configuration).
   b. A window is displayed with any detected debug probes, choose the probe you want to use and click **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 due to an ongoing bug with MCUXpresso debugger for NXP54018-IoT-Module).
9. When the debugger stops at the breakpoint in `main()`, go to the **Debug** menu, and choose **Go**.

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

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).

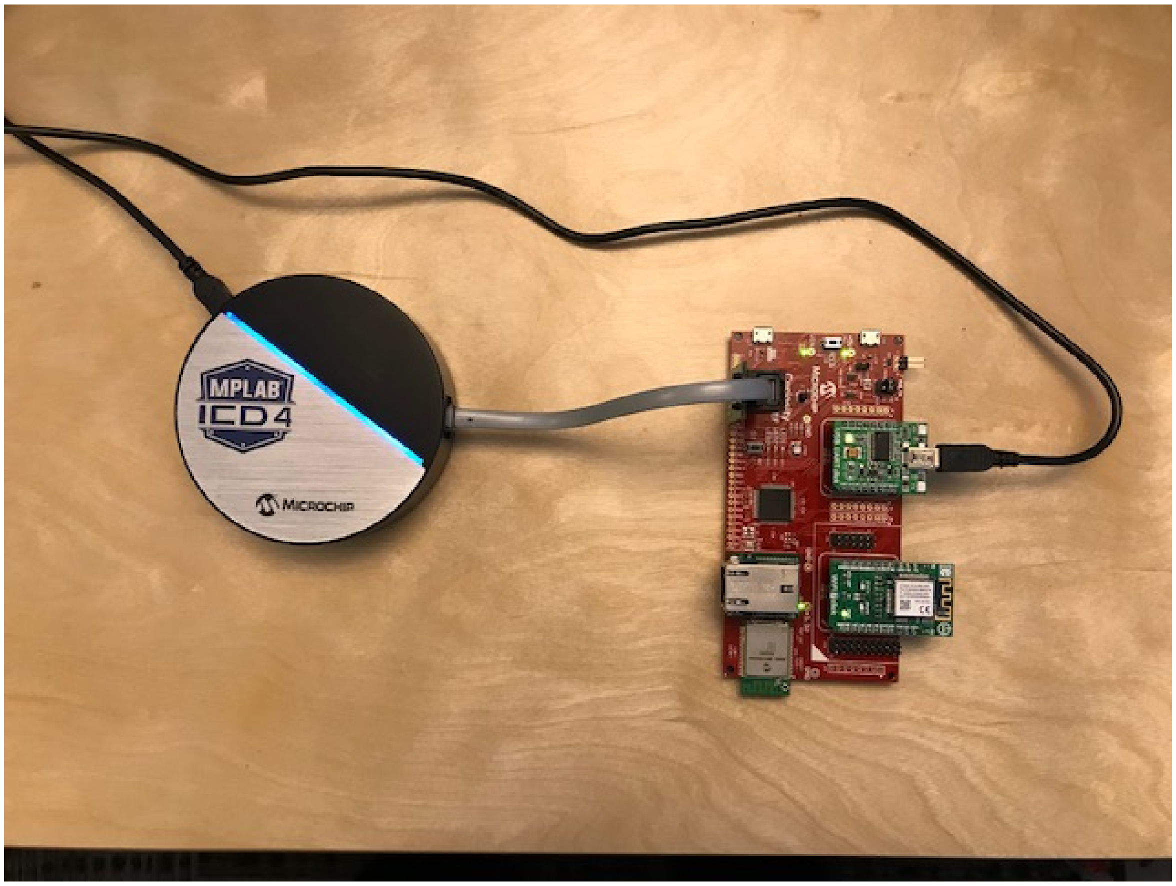
If you do not have the Microchip Curiosity PIC32MZEF bundle, you can purchase one from Microchip. You need the following items:

- 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 of the Ethernet cable 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 will turn on a solid blue when it is ready.

Setting Up Your Environment

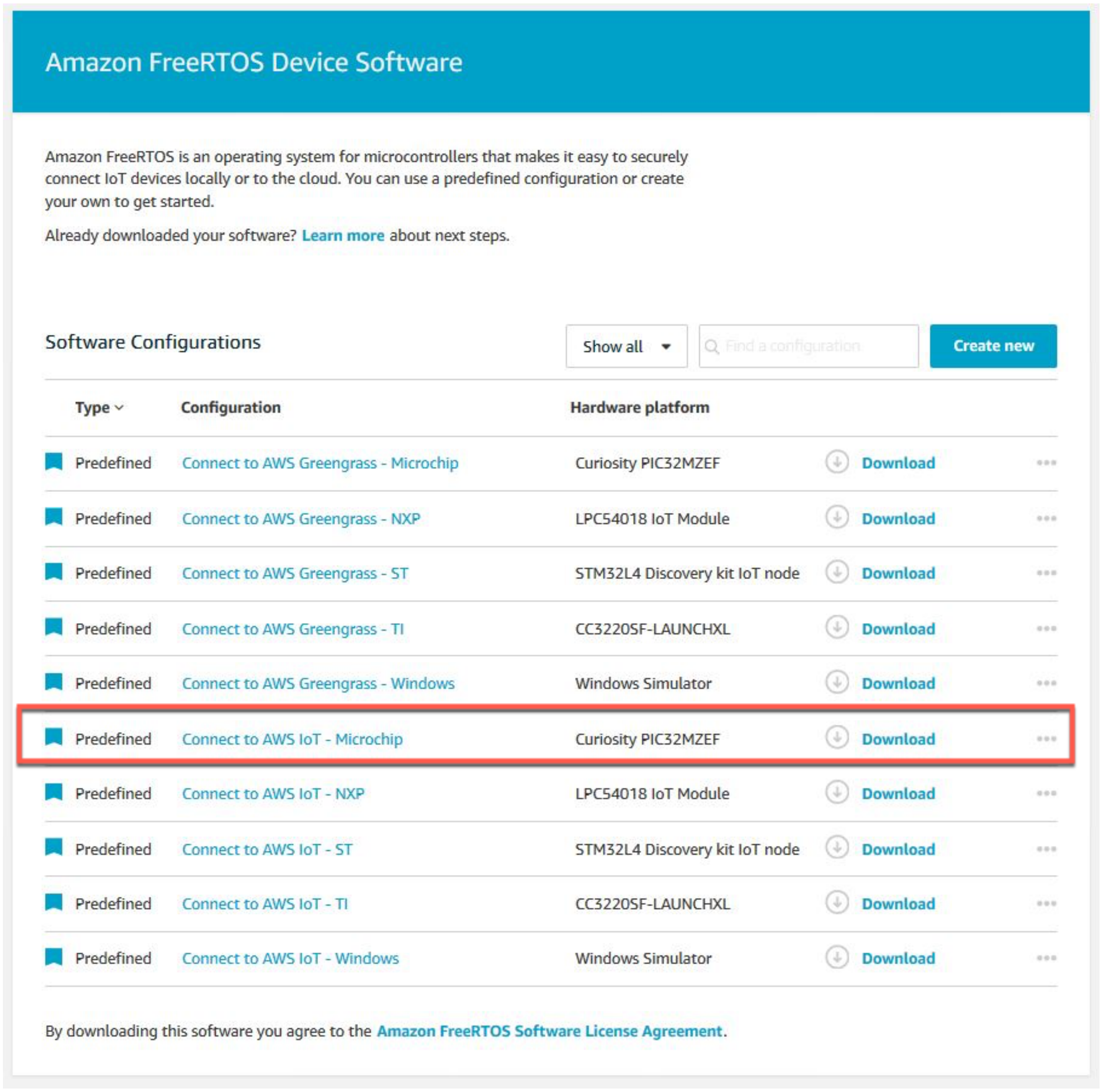
1. Install the latest Java SE SDK.
2. Install the latest version of the MPLAB X Integrated Development Environment:
   - MPLAB X Integrated Development Environment for Windows
   - MPLAB X Integrated Development Environment for macOS
   - MPLAB X Integrated Development Environment for Linux
3. Install 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
4. 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
5. 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 Build Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS and run the sample code.

Download Amazon FreeRTOS

1. Browse to the Amazon FreeRTOS page in the AWS IoT console.
2. In the left navigation pane, choose **Software**.
3. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
4. Choose **Download FreeRTOS Software**.
5. In **Software Configurations**, find **Connect to AWS IoT- Microchip**, and then choose **Download**.



Amazon FreeRTOS Device Software

Amazon FreeRTOS is an operating system for microcontrollers that makes it easy to securely connect IoT devices locally or to the cloud. You can use a predefined configuration or create your own to get started.

Already downloaded your software? Learn more about next steps.

Software Configurations

Type	Configuration	Hardware platform		
Predefined	Connect to AWS Greengrass - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS Greengrass - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS Greengrass - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS Greengrass - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS Greengrass - Windows	Windows Simulator	Download	•••
Predefined	Connect to AWS IoT - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS IoT - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS IoT - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS IoT - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS IoT - Windows	Windows Simulator	Download	•••

By downloading this software you agree to the Amazon FreeRTOS Software License Agreement.

6. Unzip the downloaded file to a folder, and make a note of the folder path. In this tutorial, this folder is referred to as `BASE_FOLDER`.

   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 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.

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>\AmazonFreeRTOS\demos\microchip\curiosity_pic32mzef\mplab`.

3. Choose **Open project** to import the project.

   Note
   You may see some warning messages when opening the project for the first time. These messages may look 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.
   ```

   These warnings may be ignored.

Configure Your Project

To configure your project, you need to know your AWS IoT endpoint.

To find your AWS IoT endpoint

1. Browse to the AWS IoT console.
2. In the left navigation pane, choose **Settings**.
3. Copy your AWS IoT endpoint from the **Endpoint** text box. It should look like `<1234567890123>.iot.<us-east-1>.amazonaws.com`.

You also need to know your Wi-FI network SSID, password, and security type and the name of the AWS IoT thing that represents your device.

In the **Project Explorer** window, open `aws_demos\application_code\common_demos\include\aws_clientcredential.h`.

Specify values for the following `#define` constants:

- `clientcredentialMQTT_BROKER_ENDPOINT`: Your AWS IoT endpoint.
- `clientcredentialIOT_THING_NAME`: The AWS IoT thing for your board.
- `clientcredentialWIFI_SSID`: The SSID for your Wi-Fi network.
- `clientcredentialWIFI_PASSWORD`: The password for your Wi-Fi network.
- `clientcredentialWIFI_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.

Make sure to save your changes.

Configure Your AWS IoT Credentials

The certificate and private key must be hard-coded into the Amazon FreeRTOS demo code. Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project.

To format your certificate and private key

1. In a browser window, open *<BASE_FOLDER>*`\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the *<ID>*`-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the *<ID>*`-private.pem.key` 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 stub file in the directory.

Run the FreeRTOS Samples

1. On the **Projects** tab, right-click the aws_demos top-level folder and then choose **Debug**.
2. The first time you debug the samples, an **ICD 4 not Found** dialog box appears. In the tree view, under the **ICD 4** node, choose the ICD4 serial number, and then choose **OK**.

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.

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

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 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.
2. Install Microsoft Visual Studio Community 2017.
3. Make sure that you have an active hard-wired Ethernet connection.

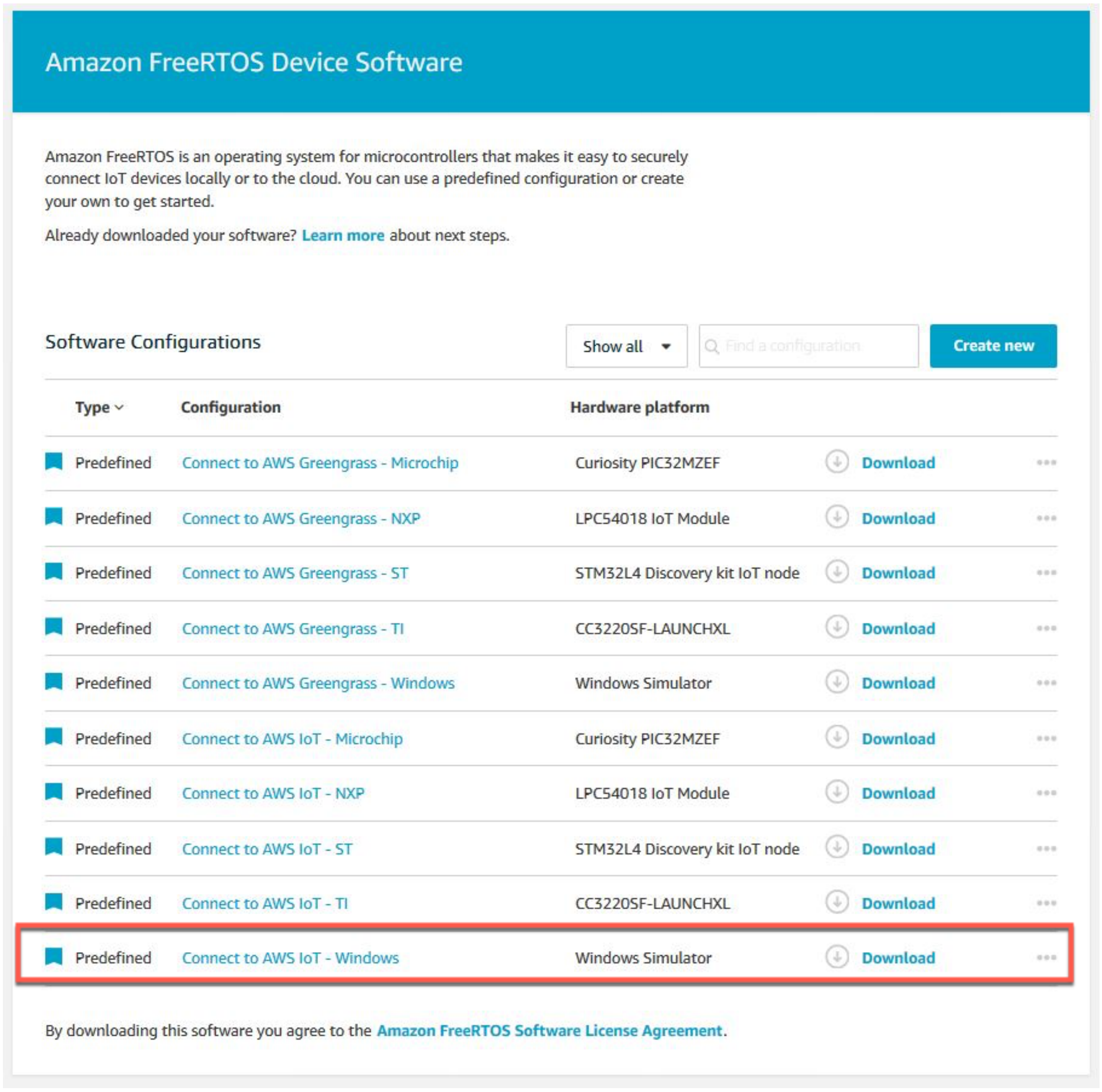
Download and Build Amazon FreeRTOS

After your environment is set up, you can download Amazon FreeRTOS and run the sample code.

Download Amazon FreeRTOS

1. In the AWS IoT console, browse to the Amazon FreeRTOS page.
2. In the left 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**.



Amazon FreeRTOS Device Software

Amazon FreeRTOS is an operating system for microcontrollers that makes it easy to securely connect IoT devices locally or to the cloud. You can use a predefined configuration or create your own to get started.

Already downloaded your software? Learn more about next steps.

Software Configurations

Type	Configuration	Hardware platform		
Predefined	Connect to AWS Greengrass - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS Greengrass - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS Greengrass - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS Greengrass - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS Greengrass - Windows	Windows Simulator	Download	•••
Predefined	Connect to AWS IoT - Microchip	Curiosity PIC32MZEF	Download	•••
Predefined	Connect to AWS IoT - NXP	LPC54018 IoT Module	Download	•••
Predefined	Connect to AWS IoT - ST	STM32L4 Discovery kit IoT node	Download	•••
Predefined	Connect to AWS IoT - TI	CC3220SF-LAUNCHXL	Download	•••
Predefined	Connect to AWS IoT - Windows	Windows Simulator	Download	•••

By downloading this software you agree to the Amazon FreeRTOS Software License Agreement.

6. Unzip the downloaded file to a folder, and make a note of the folder path. In this tutorial, this folder is referred to as `BASE_FOLDER`.

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 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.

Load the Amazon FreeRTOS Sample Code into Visual Studio

1. In Visual Studio, go to the **File** menu, choose **Open**, choose **File/Solution**, navigate to *`<BASE_FOLDER>`*`\AmazonFreeRTOS\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.

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.

   You can close the application window after you have identified the number for your hard-wired Ethernet interface.
2. Open `AmazonFreeRTOS\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.

Configure Your AWS IoT Endpoint

You must specify a custom AWS IoT endpoint for the FreeRTOS sample code to connect to AWS IoT.

1. Browse to the AWS IoT console.
2. In the left navigation pane, choose **Settings**.
3. Copy your custom AWS IoT endpoint from the **Endpoint** text box. It should look like `<c3p0r2d2a1b2c3>.iot.<us-east-1>.amazonaws.com`.
4. Open `aws_demos/application_code/common_demos/include/aws_clientcredential.h` and set `clientcredentialMQTT_BROKER_ENDPOINT` to your AWS IoT endpoint.

Configure Your AWS IoT Credentials

The certificate and private key must be hard-coded into the Amazon FreeRTOS simulator code. Amazon FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project.

To format your certificate and private key

1. In a browser window, open `<BASE_FOLDER>\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the `<ID>-certificate.pem.crt` you downloaded from the AWS IoT console.

3. Under **Private Key PEM file**, choose the `<ID>-private.pem.key` you downloaded from the AWS IoT console.
4. Choose **Generate and save aws_clientcredential_keys.h** and save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the stub file in the directory.

Run the 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 left navigation pane, choose **Test** to open the MQTT client.
4. In the **Subscription topic** text box, type `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.

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

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 Secure Sockets.

`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.

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",
      "Resource": "arn:aws:iot:us-west-2:123456789012:topicfilter/$aws/things/
thingName/shadow/*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource":
      "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/thingName/shadow/*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource":
      "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/thingName/shadow/*"
    }
  ]
}
```

2. Uncomment the declaration of and call to `vStartShadowDemoTasks` in `aws_demo_runner.c`. This function creates a task that runs the `prvShadowMainTask` 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 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.

- Getting Started with the Texas Instruments CC3220SF-LAUNCHXL (p. 7)
- Getting Started with the STMicroelectronics STM32L4 Discovery Kit IoT Node (p. 12)
- Getting Started with the NXP LPC54018 IoT Module (p. 15)
- Getting Started with the Microchip Curiosity PIC32MZEF (p. 19)
- Getting Started with the FreeRTOS Windows Simulator (p. 25)

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.

This section walks you through using the OTA demo application to perform an over-the-air (OTA) update.

Before you create an OTA update, complete the Getting Started with Amazon FreeRTOS (p. 4) tutorial to make sure you can connect your device to AWS IoT.

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.

OTA Update Prerequisites

Create an Amazon S3 Bucket to Store a Firmware Image

Firmware updates are stored in Amazon S3 buckets.

To create an Amazon S3 bucket

1. Go to the https://console.aws.amazon.com/s3/.
2. Choose **Create bucket**.
3. Type a bucket name, and then choose **Next**.
4. On the **Create bucket** page, choose **Versioning**.
5. Choose **Enable versioning**, and then choose **Save**.
6. Choose **Next**.
7. Choose **Next** to accept the default permissions.
8. Choose **Create bucket**.

Grant OTA Update Permissions to Your AWS User Account

To perform OTA updates, you must define:

- **An OTA service role**.

  The OTA Update service assumes this role to create and manage OTA update jobs on your behalf.
- **An OTA user policy**.

  This policy grants your IAM user access to a number of services required for OTA updates.
- **A code signing policy**.

  The code signing policy grants your IAM user access to the Code Signing for Amazon FreeRTOS service.

Creating an OTA Update Service Role

To create an OTA service role

1. Sign in to the https://console.aws.amazon.com/iam/.
2. From the navigation pane, choose **Roles**.
3. Choose the **Create role** button.
4. Under **Select type of trusted entity**, choose **AWS Service**.
5. Choose **IoT** from the list of AWS services.
6. Choose **Next: Permissions**.
7. Choose **Next: Review**.
8. Type a role name and description, and then choose **Create role**.

To add OTA update permissions to your OTA service role

1. In the search box on the IAM console page, type your role's name and then choose it from the list.

2. Choose the **Attach policy** button.
3. In the **Search** box, type **AWSIoTOTAUpdate**, choose it in the list of managed policies, and then choose the **Attach policy** button.

To add Amazon S3 permissions to your OTA service role

1. In the search box on the IAM console page, type your role's name 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.

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

5. Choose **Review policy**.
6. Type a name for the policy and 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 Amazon streaming service.
- Access the Amazon FreeRTOS OTA Update service.
- Access the AWS IoT Jobs service.
- Access IAM.
- Access the Code Signing for Amazon FreeRTOS service.

Grant your IAM user the required permissions by creating an OTA user policy and attaching it to your IAM user.

To create an OTA user policy

1. Open the https://console.aws.amazon.com/iam/ console.
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 the **Create policy** button.

7. On the **JSON** tab, copy and paste the following policy document into the policy editor:

```
{
 "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:ListThings",
   "iot:ListThingGroups",
   "iot:CreateStream",
   "iot:CreateOTAUpdate",
   "iot:GetOTAUpdate",
   "iot:ListJobs",
   "iot:ListJobExecutionsForJob",
   "iot:DescribeJob",
   "iot:GetJobDocument",
   "iam:ListRoles",
   "signer:ListSigningJobs",
   "signer:StartSigningJob",
   "signer:DescribeSigningJob"
  ],
  "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 you store your OTA update firmware image. Replace *<your-account-id>* with your AWS account ID. You can find your AWS account ID in the upper-right corner 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.

8. Choose the **Review policy** button.
9. Type a name for your new OTA user policy, and then choose the **Create policy** button.

To attach the OTA user policy to your IAM user

1. In the IAM console, on 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 the **Next: Review** button.
6. Choose the **Add permissions** button.

Granting Access to the Code Signing Service

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 use the Code Signing for Amazon FreeRTOS service in the AWS IoT console to sign your code. To sign your code automatically, you must grant your AWS account access to the Code Signing for Amazon FreeRTOS service.

To grant your AWS account access to Code Signing for Amazon FreeRTOS

1. Sign in to the https://console.aws.amazon.com/iam/.
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.

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

5. Choose **Review policy**.
6. Type 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**.

Command Line Tools

Make sure you have the most recent versions of openssl and the AWS CLI installed.

Using the OTA Update Demo Application on the TI CC3200SF-LAUNCHXL

This section will walk you through using the OTA Update demo application on the Texas Instruments CC3220SF-LAUNCHXL. You will create a code-signing certificate, install an initial version of the demo application (firmware), update the version of the firmware, upload the new version of the firmware and then create an OTA update job that will perform the over the air update on your Texas Instruments CC3220SF-LAUNCHXL.

Create a Code-Signing Certificate and Private Key

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).

Creating Code-Signing Certificates for Texas Instruments CC3220SF-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 1.40.01.00. By default, the files you need are located here:

`C:\ti\simplelink_cc32xx_sdk_1_40_01_00\tools\cc32xx_tools\certificate-playground` (Windows)

`/Applications/Ti/simplelink_cc32xx_sdk_1_40_01_00/tools/cc32xx_tools/certificate-playground` (macOS)

The certificates in the SimpleLink CC3220 SDK are in DER format. You need to convert them to PEM format to create a self-signed code-signing certificate.

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_1_40_01_00\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert-key` (Windows)

   `/Applications/Ti/simplelink_cc32xx_sdk_1_40_01_00/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_1_40_01_00\tools\cc32xx_tools\certificate-playground/dummy-root-ca-cert` (Windows)

   `/Applications/Ti/simplelink_cc32xx_sdk_1_40_01_00/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:

   ```
   aws acm import-certificate --certificate file://tisigner.crt.pem --private-key file://
   tisigner.key --certificate-chain file://dummy-root-ca-cert.pem
   ```

   This command displays an ARN for your certificate. You need this ARN when you create an OTA update job.

   Note
   This step assumes you are going to use the Code Signing for Amazon FreeRTOS service to sign your firmware images. Although the use of the service is recommended, you can sign your firmware images manually.

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. In these steps, we use the OTA demo application for both the initial firmware and the firmware update.

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.

   Note
   The OTA Updates feature is currently in beta.
4. Under **Libraries**, choose **Add another library**, and then choose **OTA Updates**.
5. Under **Name required**, type **`Connect-to-IoT-OTA-TI`**, and then choose **Create and download**.

To build the demo application

1. Extract the .zip file.
2. Follow the instructions in Getting Started with Amazon FreeRTOS (p. 4), 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 `aws_demos/application_code/source/aws_demo_runner.c` and comment out the call to `vStartMQTTEchoDemo()` and its `extern` declaration. Uncomment the call to `vStartOTAUpdateDemoTask()` and its `extern` declaration.
4. Build the project and make sure it compiles without errors.
5. Run a terminal emulator and connect to your board using the following settings:
   - Baud rate: 115200
   - Data bits: 8
   - 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.

The terminal emulator should display text like the following:

```
Device came up in Station mode

 Device disconnected from the AP on an ERROR..!!

 [WLAN EVENT] STA Connected to the AP: Guest , BSSID:
      <00:00:aa:bb:cc:dd>

 [NETAPP EVENT] IP acquired by the device


 Device has connected to <SSID>

 Device IP Address is <ip-address>


 6 2514 [OTA] OTA demo version 0.9.0
 7 2514 [OTA] Creating MQTT Client...
 8 2514 [OTA] Connecting to broker...
 9 2514 [OTA] Sending command to MQTT task.
 10 2514 [MQTT] Received message 10000 from queue.
```

To burn the demo application onto your board

1. On your Texas Instruments developer board, 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**, and then choose **Start Image Creator**.
4. Choose **New Project**.
5. On the **Start new project** page, type a name for your project. Set **Device Type** to **CC3220SF**, set **Device Mode** to **Develop**, and then choose **Create Project**.
6. Disconnect your terminal emulator and choose the **Connect** button on the right side of the Uniflash application window.
7. On the left, 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_1_40_01_00/tools/cc32xx_tools/certificate-playground` directory, select `dummy-root-ca-cert`, choose **Open**, and then choose **Write**.
10. 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**.
11. 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**.
12. In the file dialog box, confirm the file name is set to **mcuflashimg.bin**. Select the **Vendor** check box. Under **File Token**, type **`1952007250`**. Under **Private Key File Name**, choose **Browse** and then choose `tisignerkey` from the working directory where you created the code-signing certificate and private key. Under **Certification File Name**, choose `tisigner.crt.der`, and then choose **Write**.
13. In the left pane, under **Files**, select **Trusted Root-Certificate Catalog**.
14. Clear the **Use default Trusted Root-Certificate Catalog** check box.
15. Under **Source File**, choose **Browse**, select **simplelink_cc32xx_sdk_1_40_01_00/tools/certificate-playground\certcatalogPlayGround20160911.lst**, and then choose **Open**.
16. Under **Signature Source File**, choose **Browse**, select **simplelink_cc32xx_sdk_1_40_01_00/tools/certificate-playground\certcatalogPlayGround20160911.lst.signed.bin**, and then choose **Open**.
17. Choose the button to save your project.
18. Choose the button.
19. Choose **Program Image (Create and Program)**.
20. After 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.

Creating a Firmware Update

Now you can create an updated firmware image to install using the OTA Update service. In these steps, we just update the firmware version of the OTA demo application. Because the OTA demo displays the version of the firmware on device reset, we can use the displayed version to verify the update was successful.

To create an updated version of the firmware

1. Open the `aws_demos` project in Code Composer Studio.
2. Open `aws_demos/application_code/common_demos/include/aws_application_version.h` and set `APP_VERSION_BUILD` to 1.
3. Rebuild the `aws_demos` project.

Uploading Updated Firmware

The Amazon FreeRTOS OTA Update service loads your updated firmware from an Amazon S3 bucket.

To upload your firmware image to an Amazon S3 bucket

1. Go to the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. In **Search for buckets**, type your bucket name, and then choose it from the list. If you haven't created a bucket, see Create an Amazon S3 Bucket to Store a Firmware Image (p. 33).
3. Choose **Upload**.
4. Choose **Add files**, choose the firmware image you want to upload, and then choose **Next**.
5. Choose **Next**, and then choose **Next** again.
6. Choose **Upload**.

Create an OTA Update Job

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. Under **Select devices to update**, choose **Select**, and then choose the **Things** tab in the device list.
5. Select the check box next to the IoT thing associated with your CC3220SF-LAUNCHXL board.
6. Under **Select and sign your firmware image**, choose **Sign a new firmware image for me**.
7. Under **Pathname of code signing certificate on device**, type **`tisigner.crt.der`**
8. Make sure **Pathname of firmware image on device** is set to **`/sys/mcuflashimg.bin`** and **Device hardware platform** is set to **`CC3220SF-LAUNCHXL`**.
9. Under **Select your firmware image in S3**, choose **Select**, and then choose the aws_demos.bin image that contains your updated image.
10. Under **Code signing certificate**, choose **Select**, and then choose your code-signing certificate. Code-signing certificates are listed by ARN. The ARN for your certificate was displayed when you registered it with ACM.
11. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.
12. Under **IAM role for OTA update job**, choose your OTA service role.
13. Type an alphanumeric ID for your job.
14. Choose **Create**.

The job appears with a status of **IN PROGRESS** in the AWS IoT console window.

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 CC3220SF-LAUNCHXL. You should see output that indicates the device is downloading the updated firmware:

```
678 603530 [OTA] [OTA] Queued: 2   Processed: 2   Dropped: 0
 679 603959 [OTA Task] [OTA] Set job doc parameter [ jobId:
 89cb1be6_e6fa_4260_99cc_54569d9e87dd ]
 680 603960 [OTA Task] [OTA] Set job doc parameter [ streamname: 353c253f-5a06-43b2-99d6-
f6041a266db8 ]
 681 603960 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
 682 603960 [OTA Task] [OTA] Set job doc parameter [ filesize: 129132 ]
 683 603961 [OTA Task] [OTA] Set job doc parameter [ fileid: 0 ]
 684 603961 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
 685 603961 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
 686 603963 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa: bf/
ET94tx3PULTzI4VNFxdSDapeK65dn ]
 687 603963 [OTA Task] [OTA] Job was accepted. Attempting to start transfer.
 688 603963 [OTA Task] Sending command to MQTT task.
 689 603963 [MQTT] Received message 80000 from queue.
 690 604065 [MQTT] MQTT Subscribe was accepted. Subscribed.
 691 604065 [MQTT] Notifying task.
 692 604066 [OTA Task] Command sent to MQTT task passed.
 693 604066 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI_CCS3200SF/
streams/353c253f-5a06-43b2-99d6-f6041a266db8

 694 604530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 695 605530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 696 606530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 697 607530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 698 608530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 699 609530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 700 610530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 701 611530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 702 611776 [OTA Task] [OTA] file token: 74594452
 703 612530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 704 613530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 705 614530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 706 615530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 707 616530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 708 617530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 709 618530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 710 619530 [OTA] [OTA] Queued: 1   Processed: 0   Dropped: 0
 711 620047 [OTA Task] [OTA] file ready for access.
 712 620047 [OTA Task] [OTA] Returned buffer to MQTT Client.
 713 620048 [OTA Task] Sending command to MQTT task.
 714 620048 [MQTT] Received message 90000 from queue.
 715 620048 [MQTT] Notifying task.
 716 620049 [OTA Task] Command sent to MQTT task passed.
 717 620049 [OTA Task] [OTA] Published file request to $aws/bin/things/TI_CCS3200SF/
streams/353c253f-5a06-43b2-99d6-f6041a266db8718 620530 [OTA] [OTA] Queued: 3   Processed: 3
   Dropped: 0
 719 620653 [OTA Task] [OTA] Received file block 0, size 1024
 720 620667 [OTA Task] [OTA] Remaining: 126
 721 620668 [OTA Task] [OTA] Returned buffer to MQTT Client.
 722 620669 [OTA Task] [OTA] Received file block 1, size 1024
 723 620697 [OTA Task] [OTA] Remaining: 125
 ...
```

After the device downloads the updated firmware, it restarts and then installs the firmware. You can see what's happening in the UART terminal:

```
1285 756833 [OTA Task] [OTA] Resetting MCU to activate new image.
 0 0 [Tmr Svc] Simple Link task created

 Device came up in Station mode

 1 89 [Tmr Svc] Starting key provisioning...
 2 89 [Tmr Svc] Write root certificate...
 3 190 [Tmr Svc] Write device private key...
 4 290 [Tmr Svc] Write device certificate...
 SL Disconnect...

 5 387 [Tmr Svc] Key provisioning done...
 Device came up in Station mode

 [WLAN EVENT] STA Connected to the AP: Guest , BSSID: 44:48:c1:ba:b2:c3

 [NETAPP EVENT] IP acquired by the device


 Device has connected to Guest

 Device IP Address is 192.168.14.217


 6 2440 [OTA] OTA demo version 0.9.3
 7 2440 [OTA] Creating MQTT Client...
 8 2440 [OTA] Connecting to broker...
 9 2440 [OTA] Sending command to MQTT task.
 10 2440 [MQTT] Received message 10000 from queue.
 ...
```

Using the OTA Update Demo Application on the Microchip Curiosity PIC32MZEF

This section will walk you through using the OTA Update demo application on the Microchip Curiosity PIC32MZEF. You will create a code-signing certificate, install an initial version of the demo application (firmware), update the version of the firmware, upload the new version of the firmware and then create an OTA update job that will perform the over the air update on your Microchip Curiosity PIC32MZEF.

Creating Code-Signing Certificates

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 certificates to the 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. For example:

```
{
    "CertificateArn": "arn:aws:acm:us-
west-2:123123123123:certificate/23312627-963b-4798-888b-09e89071a861"
}
```

   You will need to specify this ARN when you create an OTA update job in the AWS IoT console.

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. The OTA demo application contains the code for the initial firmware and the code to listen for firmware updates.

This tutorial assumes you have followed the instructions in:

1. Prerequisites (p. 4)
2. Setting Up the Microchip Curiosity PIC32MZEF Hardware (p. 20)
3. Setting Up Your Environment (p. 21)

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.

   Note
   The OTA Updates feature is currently in beta.
4. Under **Libraries**, choose **Add another library**, and then choose **OTA Updates**.
5. Under **Demo projects** choose **OTA Update**.
6. Under **Name required** type 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. Build the project and make sure it compiles without errors.
4. Start a terminal emulator and connect to your board using the following settings:
   - Baud rate: 115200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
5. 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 and you should see the following output.

```
*****************************************************


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 = 0x1d08f7ff
configuration memory
boot config memory

Programming/Verify complete

Running
```

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] vDHCPProcess: offer c0a804beip
3 7230 [IP-task] vDHCPProcess: 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.
34 10336 [MQTT] Notifying task.
35 10336 [OTA Task] Command sent to MQTT task passed.
36 10336 [OTA Task] [OTA] Subscribed to topic: $aws/things/Microchip/jobs/notify-next

37 10336 [OTA Task] [OTA] Check For Update #0
38 10336 [OTA Task] Sending command to MQTT task.
39 10336 [MQTT] Received message 40000 from queue.
40 10366 [IP-task] Socket sending wakeup to MQTT task.
```

```
41 10366 [MQTT] Received message 0 from queue.
42 10366 [MQTT] MQTT Publish was successful.
43 10366 [MQTT] Notifying task.
44 10366 [OTA Task] Command sent to MQTT task passed.
45 10376 [IP-task] Socket sending wakeup to MQTT task.
46 10376 [MQTT] Received message 0 from queue.
47 10376 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:Microchip ]
48 10376 [OTA Task] [OTA] Missing job parameter: execution
49 10376 [OTA Task] [OTA] Missing job parameter: jobId
50 10376 [OTA Task] [OTA] Missing job parameter: jobDocument
51 10378 [OTA Task] [OTA] Missing job parameter: ts_ota
52 10378 [OTA Task] [OTA] Missing job parameter: files
53 10378 [OTA Task] [OTA] Missing job parameter: streamname
54 10378 [OTA Task] [OTA] Missing job parameter: certfile
55 10378 [OTA Task] [OTA] Missing job parameter: filepath
56 10378 [OTA Task] [OTA] Missing job parameter: filesize
57 10378 [OTA Task] [OTA] Missing job parameter: sig-sha256-ecdsa
58 10378 [OTA Task] [OTA] Missing job parameter: fileid
59 10378 [OTA Task] [OTA] Missing job parameter: attr
60 10378 [OTA Task] [OTA] Returned buffer to MQTT Client.
61 11367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
62 12367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
63 13367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
64 14367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
65 15367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
66 16367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
```

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.

To burn the demo application onto your board

1. Open `aws_demos/lib/aws/ota/aws_codesign_keys.h`.
2. Paste the contents of "ecdsasigner.crt" created earlier into the `static const char signingcredentialSIGNING_CERTIFICATE_PEM` variable. Following the same format as aws_clientcredential_keys.h, each line must end with the new line character ('\n') and be surrounded by quotes.

   For example, your certificate should look similar to the following:

```
"-----BEGIN CERTIFICATE-----\n"
"MIIBXTCCAQOgAwIBAgIJAM4DeybZcTwKMAoGCCqGSM49BAMCMCExHzAdBgNVBAMM\n"
"FnRlc3Rf62lnbmVyQGFtYXpvbi5jb20wHhcNMTcxMTAzMTkxODM1WhcNMTgxMTAz\n"
"MTkxODM2WjAhMR8wHQYDVQBBZZZ0ZXN0X3NpZ25lckBhbWF6b24uY29tMFkwEwYH\n"
"KoZIzj0CAQYIKoZIzj0DAQcDQgAERavZfvwL1X+E4dIF7dbkVMUn4IrJ1CAsFkc8\n"
"gZxPzn683H40XMKltDZPEwr9ng78w9+QYQg7ygnr2stz8yhh06MkMCIwCwYDVR0P\n"
"BAQDAgeAMBMGA1UdJQQMMAoGCCsGAQUFBwMDMAoGCCqGSM49BAMCA0gAMEUCIF0R\n"
"r5cb7rEUNtWOvGd05MacrgOABfSoVYvBOK9fP63WAqt5h3BaS123coKSGg84twlq\n"
"TkO/pV/xEmyZmZdV+HxV/OM=\n"
"-----END CERTIFICATE-----\n";
```

3. Paste the contents of "ecdsasigner.key" created earlier into the `static const char signingcredentialSIGNING_PRIVATE_KEY_PEM` variable. Following the same format as aws_clientcredential_keys.h, each line must end with the new line character ('\n') and be surrounded by quotes.
4. Rebuild the aws_demos project and make sure it compiles without errors.
5. Click on the from the top tool bar.

6. 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.

Creating a Firmware Update

Now you can create an updated firmware image to install using the OTA Update service. In these steps, we just update the firmware version of the OTA demo application. Because the OTA demo displays the version of the firmware on device reset, we can use the displayed version to verify the update was successful.

To create an updated version of the firmware

1. Open the `aws_demos` project in the MPLAB X IDE.
2. Open `aws_demos/application_code/common_demos/include/aws_application_version.h` and set `APP_VERSION_BUILD` to 1.
3. Rebuild the `aws_demos` project.

Uploading Updated Firmware

The Amazon FreeRTOS OTA Update service loads your updated firmware from an Amazon S3 bucket.

To upload your firmware image to an Amazon S3 bucket

1. Go to the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. In **Search for buckets**, type your bucket name, and then choose it from the list. If you haven't created a bucket, see Create an Amazon S3 Bucket to Store a Firmware Image (p. 33).
3. Choose **Upload**.
4. Choose **Add files**, choose the firmware image you want to upload. In this case, choose `AmazonFreeRTOS\demos\microchip\curiosity_pic32mzef\mplab\dist\pic32mz_ef_curiosity\productionmplab.production.bin` and then choose **Open**.
5. Choose **Next**, and then choose **Next** again.
6. Choose **Upload**.

Creating an OTA Update Job

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. Under **Select devices to update**, choose **Select**, and then choose the **Things** tab in the device list.
5. Select the check box next to the IoT thing associated with your Microchip Curiosity PIC32MZEF board.
6. Under **Select and sign your firmware image**, choose **Sign a new firmware image for me**.
7. Choose **Curiosity PIC32MZEF** in the **Device hardware platform** drop down selection.
8. Under **Pathname of firmware image on device**, type any alphanumeric dummy string. This field cannot be blank, but is not used by the Microchip Curiosity PIC32MZEF.
9. Under **Pathname of code signing certificate on device**, type any alphanumeric dummy string. This field cannot be blank, but is not used by the Microchip Curiosity PIC32MZEF.

10. Under **Select your firmware image in S3**, choose **Select**, and then choose the mplab.production.bin image that contains your updated image.
11. Under **Code signing certificate**, choose **Select**, and then choose your code-signing certificate. Code-signing certificates are listed by ARN. The ARN for your certificate was displayed when you registered it with ACM.
12. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.
13. Under **IAM role for OTA update job**, choose your OTA service role.
14. Type an alphanumeric ID for your job.
15. Choose **Create**.

The job appears with a status of **IN PROGRESS** in the AWS IoT console window.

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 Microchip Curiosity PIC32MZEF. You should see output that indicates the device is downloading the updated firmware:

```
101 23773 [OTA Task] [OTA] Set job doc parameter [ jobId:
 42203c2a_39e7_4651_a791_9aba2a965656 ]
102 23773 [OTA Task] [OTA] Set job doc parameter [ streamname:
 f9d9eff7-51b5-4442-840c-8dfe73366ed9 ]
103 23773 [OTA Task] [OTA] Set job doc parameter [ filepath: dummy ]
104 23773 [OTA Task] [OTA] Set job doc parameter [ filesize: 588156 ]
105 23775 [OTA Task] [OTA] Set job doc parameter [ fileid: 0 ]
106 23775 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
107 23775 [OTA Task] [OTA] Set job doc parameter [ certfile: dummy ]
108 23775 [OTA Task] [OTA] Set job doc parameter [ sig-sha256-ecdsa: MEUCIBvcJ50rrQGdB
+8d4vPdoBKcCECS ]
109 23775 [OTA Task] [OTA] Job was accepted. Attempting to start transfer.
110 23775 [OTA Task] Sending command to MQTT task.
111 23775 [MQTT] Received message 80000 from queue.
112 23852 [IP-task] Socket sending wakeup to MQTT task.
113 23852 [MQTT] Received message 0 from queue.
114 23901 [IP-task] Socket sending wakeup to MQTT task.
115 23901 [MQTT] Received message 0 from queue.
116 23922 [IP-task] Socket sending wakeup to MQTT task.
117 23922 [MQTT] Received message 0 from queue.
118 23922 [MQTT] MQTT Subscribe was accepted. Subscribed.
119 23922 [MQTT] Notifying task.
120 23923 [OTA Task] Command sent to MQTT task passed.
121 23923 [OTA Task] [OTA] Subscribed to topic: $aws/things/MchpBoard-1/streams/
f9d9eff7-51b5-4442-840c-8dfe73366ed9
122 23926 [OTA Task] [OTA-MCHP] Create - Erased the flash OK
123 23926 [OTA Task] [OTA] Returned buffer to MQTT Client.
124 24241 [OTA] [OTA] Queued: 2   Processed: 2   Dropped: 0
125 25241 [OTA] [OTA] Queued: 2   Processed: 2   Dropped: 0
126 25923 [OTA Task] Sending command to MQTT task.
127 25923 [MQTT] Received message 90000 from queue.
128 25923 [MQTT] Notifying task.
129 25924 [OTA Task] Command sent to MQTT task passed.
130 25925 [OTA Task] [OTA] Published file request to $aws/bin/things/MchpBoard-1/streams/
f9d9eff7-51b5-4442-840c-8dfe73366ed9/get131 26061 [IP-task] Socket sending wakeup to MQTT
 task.
132 26061 [MQTT] Received message 0 from queue.
133 26241 [OTA] [OTA] Queued: 2   Processed: 2   Dropped: 0
134 26423 [IP-task] Socket sending wakeup to MQTT task.
135 26423 [MQTT] Received message 0 from queue.
136 26434 [IP-task] Socket sending wakeup to MQTT task.
```

```
137 26434 [MQTT] Received message 0 from queue.
138 26437 [OTA Task] [OTA] Received file block 0, size 1024
139 26449 [OTA Task] [OTA] Remaining: 574
140 26449 [OTA Task] [OTA] Returned buffer to MQTT Client.
141 26485 [IP-task] Socket sending wakeup to MQTT task.
142 26485 [MQTT] Received message 0 from queue.
143 26496 [IP-task] Socket sending wakeup to MQTT task.
144 26496 [MQTT] Received message 0 from queue.
145 26499 [OTA Task] [OTA] Received file block 1, size 1024
146 26511 [OTA Task] [OTA] Remaining: 573
```

After the device downloads the updated firmware, it restarts and then installs the firmware. If you are connected by a debugger to the MPLAB X IDE, then you will need to press **Continue** in the top debugging menu or F5. You can see what's happening in the UART terminal:

```
3941 68629 [OTA Task] [OTA-MCHP] Resetting MCU to activate new image.
AWS Validate: no valid signature in descr: 0xbd000000
AWS Validate: Valid CRC found in descr: 0xbd100000
AWS Validate: valid signature found in upper


>AWS Launch:  Map PFM Up to Low: 0xbd100000
AWS Launch:  wait for app at: 0x9d000020
WILC1000: Initializing...
0 0

>[None] Seed for randomizer: 3644421320
1 0 [None] Random numbers: 000050CA 00000EB8 00002B3A 0000290E
Chip ID 1503a0
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 7238 [IP-task] vDHCPProcess: offer c0a82b6fip
3 7269 [IP-task] vDHCPProcess: offer c0a82b6fip
4 7269 [IP-task]

IP Address: 192.168.43.111
5 7269 [IP-task] Subnet Mask: 255.255.255.0
6 7269 [IP-task] Gateway Address: 192.168.43.1
7 7269 [IP-task] DNS Server Address: 192.168.43.1


8 7271 [OTA] OTA demo version 0.9.1
```

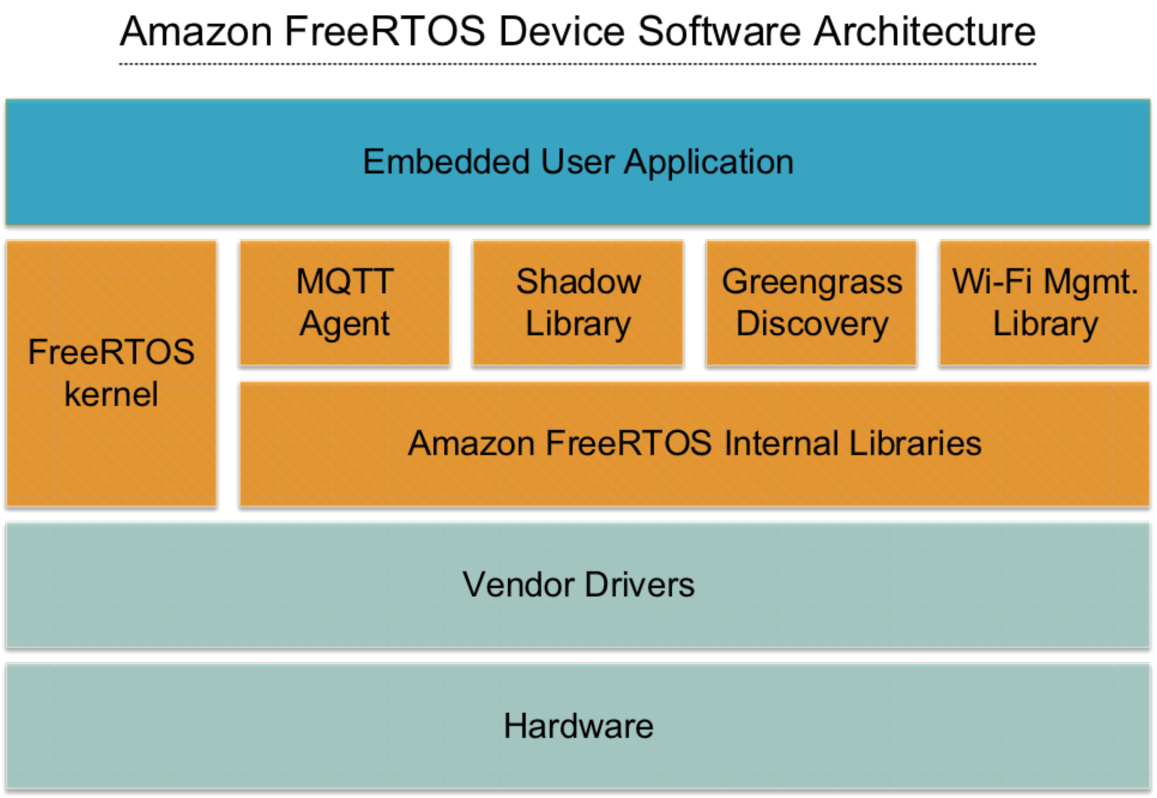
Amazon FreeRTOS Developer Guide

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

Topics

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 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.


Amazon FreeRTOS Device Software Architecture
Embedded User Application
FreeRTOS kernel
MQTT Agent
Shadow Library
Greengrass Discovery
Wi-Fi Mgmt. Library
Amazon FreeRTOS Internal Libraries
Vendor Drivers
Hardware

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).
- Inter-task 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 inter-task 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 within 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

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`

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`

Similar to heap_4. Can span the heap across multiple, non-adjacent memory areas.

Inter-task Coordination

Queues

Queues are the primary form of inter-task communication. They can be used to send messages between tasks and between interrupts and tasks. In most cases, they are used as thread-safe FIFO (First In First Out) 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 `xStreamBufferSendCompletedFromISR()` API to check, and if necessary unblock, a task that is waiting for the data.

Receiving Data

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

`xStreamBufferReceive()` allows a block time to be specified. If `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 `xStreamBufferCreate()` is called. It can be changed by calling `xStreamBufferSetTriggerLevel()`.

`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.

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 `<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 `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.

Sending Data

`xMessageBufferSend()` is used to send data to a message buffer from a task. `xMessageBufferSendFromISR()` is used to send data to a message buffer from an interrupt service routine (ISR).

`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 users 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.

FreeRTOS Libraries

This section describes how to use the Amazon FreeRTOS libraries when you are writing embedded applications.

Topics

Cloud Connectivity

Topics

MQTT

The MQTT agent implements the MQTT protocol, which is a lightweight protocol designed for constrained devices. 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.

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.

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 on the Internet of Things on AWS blog.

Device Shadows

The Amazon FreeRTOS API provides functions to create, delete, and update a device's shadow. For more information, 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 Amazon FreeRTOS device shadow APIs define functions to create, update, and delete device shadows.

Prerequisites For Using the Device Shadows API

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.

`clientcredentialCLIENT_CERTIFICATE_PEM`

The certificate PEM associated with your IoT thing. For more information, see Configure Your AWS IoT Credentials (p. 15).

`clientcredentialCLIENT_PRIVATE_KEY_PEM`

The private key PEM associated with your IoT thing. For more information, see Configure Your AWS IoT Credentials (p. 15).

Make sure the Amazon FreeRTOS MQTT library is installed on your device. For more information, see MQTT (p. 57). 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.

Using the Device Shadow APIs

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.

Greengrass Connectivity

The Greengrass Discovery API is used by your microcontroller devices to discover a Greengrass core on your network. Your device can send messages to a Greengrass core after it finds the core's endpoint.

Prerequisites

To use the Greengrass Discovery API, 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.

`clientcredentialCLIENT_CERTIFICATE_PEM`

The certificate PEM associated with your thing.

`clientcredentialCLIENT_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 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.

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\`.

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. `GD_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 Security

The Amazon FreeRTOS security API allows you to create embedded applications that communicate securely. The information in this section is intended to complement the API documentation.

Secure Sockets

The Secure Sockets interface is based on the Berkeley socket interface. It is provided for the easy onboarding of software developers from various network programming backgrounds. The reference implementation for Secure Sockets supports TLS and TCP/IP over Ethernet and Wi-Fi. See `aws_secure_sockets.h` in the Amazon FreeRTOS source code repository.

Secure Sockets Ports

This section contains information about Secure Socket ports for Amazon FreeRTOS-qualified boards. For information about creating your own port for Amazon FreeRTOS, see the Amazon FreeRTOS Porting Guide.

STM32 IoT Discovery Kit Secure Sockets Port

- This port supports up to four sockets.
- `SOCKETS_PERIPHERAL_RESET` means that the Wi-Fi module has been reset. This occurs when the Wi-Fi module stops responding or gets out of sync with the SPI driver. Call `WiFi_ConnectAP` to reconnect to your Wi-Fi network.

`Sockets_Connect`

- `SocketsSockaddr_t` uses the `usPort` and `ulAddress` fields only. `ucLength` and `ucSocketDomain` are not used.
- Supports IPv4 only.
- Sends connection information to the Wi-Fi module only. A successful return does not guarantee that the socket was able to reach the provided IP address.

`Sockets_SetSockOpt`

For `SOCKETS_SO_SNDTIMEO` and `SOCKETS_SO_RCVTIMEO`, valid values are 0 (block forever) and 30,000 milliseconds.

SOCKETS_Shutdown

`SOCKETS_Shutdown` does not send a FIN packet, but does prevent the socket from being used for send and receive.

TI CC3220SF-LAUNCHXL Secure Sockets Port

This port supports up to 16 sockets. The sockets can be secured with TLS.

`Sockets_Connect`

- `SocketsSockaddr_t` uses the `usPort` and `ulAddress` fields only. `ucLength` and `ucSocketDomain` are not used.
- Supports IPv4 only.
- Receiving a negative error code from `SOCKETS_Connect` does not mean that the socket was closed. Applications must close sockets after they receive a negative error code.
- When using a TLS-enabled socket, sometimes a connection is made even though `SOCKETS_Connect` returned an error. This might indicate that the connection cannot be authenticated using the provided root of trust. We strongly recommend that you explicitly close the socket if a handshake-related error is returned, even if the connection is made.
- In the event of handshake error, you can get information by enabling printing or by investigating the asynchronous event handler structure set in `SimpleLinkSockEventHandler`.

`Sockets_SetSockOpt`

`SOCKETS_SO_RCVTIMEO` can be specified in 10-millisecond increments.

`SOCKETS_SO_SNDTIMEO` is not used. It might be used in future versions.

SOCKETS_Send

In the event of a TX error, you can get information by investigating the TX Failed event handler structure in `SimpleLinkSockEventHandler`.

SOCKETS_Shutdown

`SOCKETS_Shutdown` does not send a FIN packet, but does prevent the socket from being used for send and receive.

Transport Layer Security

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

Public Key Cryptography Standard #11

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 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`
- `C_CreateObject CKO_CERTIFICATE pkcs11CERTIFICATE_TYPE_USER`
- `C_CreateObject CKO_CERTIFICATE pkcs11CERTIFICATE_TYPE_ROOT`
- `C_DestroyObject`

Client Authentication

- `C_Initialize`
- `C_GetSlotList`
- `C_OpenSession`
- `C_FindObjectsInit`
- `C_FindObjects`
- `C_FindObjectsFinal`
- `C_GetAttributeValue`
- `C_FindObjectsInit`
- `C_FindObjects`
- `C_FindObjectsFinal`
- `C_GetAttributeValue`
- `C_GenerateRandom`
- `C_SignInit`
- `C_Sign`

Cleanup

- `C_CloseSession`
- `C_Finalize`

Asymmetric Cryptosystem Support

The Amazon FreeRTOS PKCS#11 reference implementation supports 2048-bit RSA (signing only) as well as 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/86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de` 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 and 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.

   ```
   {
    "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/86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de` 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`.

FreeRTOS Wi-Fi Interface

The Wi-Fi interface API provides a common abstraction layer that enables applications to communicate with the lower-level wireless stack. Wi-Fi chip sets differ in features, driver implementations, and APIs. The common Wi-Fi interface simplifies application development and porting for all supported Wi-Fi chip sets. The interface provides a primary API for managing all aspects of Wi-Fi devices.

Setup, Provisioning, and Configuration

The setup APIs provide functionality to turn on Wi-Fi by initializing the radio, peripherals, and drivers. Your application must turn on Wi-Fi by calling `WIFI_On` before calling any other API. An application can turn off WI-Fi by calling `WIFI_Off` to save power. This is useful for power-constrained devices that have intermittent connectivity. Calling `WIFI_Reset` resets the Wi-Fi radio.

The Amazon FreeRTOS demos hard code Wi-Fi credentials into the application. If you cannot connect to your Wi-FI network using these credentials, you can put your FreeRTOS device into soft Access Point (AP) mode. This allows you to connect the FreeRTOS device and configure a different set of Wi-Fi credentials (SSID, password, security type, and channel). To configure AP mode, call `WIFI_ConfigureAP`. To put your device into soft AP mode, call `WIFI_StartAP`. When your device is in soft AP mode, you can connect another device, using a web browser to your FreeRTOS device and configure the new Wi-Fi credentials. To turn off soft AP mode, call `WIFI_StopAP`.

A Wi-Fi device can be configured in a particular role at a time. Device roles like Station, Access Point, P2P can be configured by calling `WIFI_SetMode`. You can get the current mode of a Wi-Fi device by calling `WIFI_GetMode`. Switching modes by calling `WIFI_SetMode` disconnects the device, if it is already connected to a network.

Connection

A Wi-Fi device turned on and switched to Station mode is ready to connect to the network using the connectivity API. When calling the connection API, you pass network parameters like SSID, password, and security type to establish a connection. You can perform a scan operation to look for networks. The scan returns the SSID, BSSID, Channel, RSSI, and security type. The scan can be performed for hidden networks. If you find a desired network in the scan, you can connect to the network by calling and providing the network password. You can disconnect a Wi-Fi device from the network by calling `WIFI_Disconnect`.

Security

The interface API supports several security types like WEP, WPA, WPA2, and Open (no security). When a device is in the Station role, you must specify the network security type when calling the `WIFI_ConnectAP` function. When a device is in soft AP mode, the device can be configured to use any of the supported security types:

- `eWiFiSecurityOpen`
- `eWiFiSecurityWEP`
- `eWiFiSecurityWPA`
- `eWiFiSecurityWPA2`

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 API 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.

Network Utilities

The Wi-Fi API also provides utility functions described in the following table:

API	Description
`WIFI_GetIP`	Gets the IP address of a device.
`WIFI_GetHostIP`	Gets the host IP address.
`WIFI_GetMAC`	Gets the MAC address of a device.
`WIFI_Ping`	Sends a ping to a device on the network.
`WIFI_Scan`	Scans for available Wi-Fi networks.

OTA Agent Library

The OTA agent library 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.

Here is the complete OTA agent interface:

`OTA_AgentInit`

Initializes the OTA engine. The caller provides messaging protocol callbacks and a timeout.

`OTA_AgentShutdown`

Cleans up after using the OTA engine.

`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.

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. This step is not required if you are not using AWS IoT Device Management, or if you prefer an alternative implementation of MQTT. For more information, see MQTT (p. 57).
2. Initialize the OTA agent by calling `OTA_AgentInit`. Your application must define an OTA callback function and a timeout. The callback implements application-specific logic that is executed after an OTA update job is complete. 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, your job completion callback is called and determines whether the update is accepted or rejected.
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 or try again.
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.

Amazon FreeRTOS Over-the-Air Updates

The Amazon FreeRTOS Over-the-Air Update service enables you to:

- Digitally sign and encrypt firmware before 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.
- Verify the authenticity and integrity of the new firmware after it's deployed to devices.
- Monitor the progress of a deployment.
- Debug a failed deployment.

Devices communicate with the Amazon FreeRTOS OTA Update service using MQTT messages. Each device must subscribe to the appropriate topics to receive messages. MQTT messages are used to notify devices that an update is available, report the status of an update, and to stream the firmware updates. Each device has its own set of MQTT topics.

OTA Update includes the following components:

OTA Manager Service

The OTA Manager service enables users to create and manage deployments of firmware images on one or more devices or MCUs. The OTA Manager Service uses the AWS IoT Jobs service to schedule deployments.

AWS IoT Jobs Service

The AWS IoT Jobs service is a cloud-based managed service for scheduling, orchestration, notification, and status reporting of OTA updates and other remote operations on distributed fleets of small devices. To update a device, you create an OTA update job. The job specifies which devices should perform the update and where to find the firmware image, among other things. When a job is deployed to a device, a job execution is created. The job execution represents a single device applying the update. For more information, see AWS IoT Jobs.

Streaming Service

The Streaming service delivers new firmware images over MQTT to your devices. The Streaming service breaks up the firmware image into chunks and delivers each chunk as an MQTT message to the devices that are being updated. The service can redeliver blocks or a full image on request.

Code Signing for Amazon FreeRTOS Service

Code Signing for Amazon FreeRTOS is a managed AWS service that enables you to sign code that you create for any IoT device that is supported by Amazon Web Services (AWS). Code Signing is integrated with Amazon FreeRTOS and AWS Certificate Manager (ACM). Amazon FreeRTOS customers can use Code Signing to sign a code image before publishing it to a microcontroller device. You can use ACM to import a third-party code signing certificate that you can use during the signing process.

OTA Library and Agent

The OTA library allows the device developer to logically separate the application from the OTA process. The OTA library controls an OTA agent that is executed as a FreeRTOS task.

The OTA agent is responsible for:

- Downloading a new executable image from the cloud.
- Validating the image in a way that is appropriate for the application and device.
- Handling interruptions during the download.
- Managing updates that are separated into multiple sections.

The OTA agent also supports a Greengrass-mediated OTA mode for devices that are not directly connected to the cloud. In this mode, the update is downloaded by a trusted Greengrass core, which then pushes the update to Amazon FreeRTOS devices connected to it.

By automating firmware signature verification, the OTA library makes it easy for you to protect the integrity of your devices. By defining a portable abstraction layer (PAL), the OTA library minimizes the burden for onboarding new hardware to OTA-enabled applications.

Currently, OTA is supported in the following regions only:

- `us-east-1` / US East (N. Virginia)
- `us-east-2` / US East (Ohio)
- `us-west-2` / US West (Oregon)
- `eu-west-1` / EU (Ireland)
- `eu-central-1` / EU (Frankfurt)
- `eu-west-2` / EU (London)
- `ap-northeast-1` / Asia Pacific (Tokyo)
- `ap-southeast-2` / Asia Pacific (Sydney)

The region you are working in is displayed in the upper-right corner of the AWS Management Console. You can use the drop-down list to change the region. Before you create an OTA update, make sure you are working in one of these supported regions.

Note
For the Amazon FreeRTOS OTA agent to commit a firmware upgrade, the firmware image must include the OTA agent library. The firmware version must be more recent than the currently installed firmware.

OTA Workflow

This section describes the OTA update workflow. OTA updates are currently supported on the following platform. Select a platform below to find a console-based tutorial for that platform:

- Texas Instruments SimpleLink Wi-Fi CC3220SF-LAUNCHXL (p. 36).
- Microchip Curiosity PIC32MZEF (p. 43).

1. Create a self-signed certificate or purchase a code-signing certificate and a private key and import them into ACM. For more information, see Create a Code-Signing Certificate and Private Key (p. 37).
2. Deploy a device with factory-provisioned firmware (for example, v1.0). The v1.0 firmware must be configured to trust the code-signing certificate created in step 1. For more information, see Installing the Initial Firmware (p. 38).
3. When a firmware update is required, make the code changes and build the new image. For more information, see Creating a Firmware Update (p. 40).
4. Upload the new firmware image into an Amazon S3 bucket. For more information, see Uploading Updated Firmware (p. 41).
5. Digitally sign the new firmware image. You can do this step manually, or you can use the AWS IoT console which uses the Code Signing for Amazon FreeRTOS service.
6. Using the AWS IoT Device Management console, schedule an OTA update job. For more information, see Create an OTA Update Job (p. 41).
7. The Amazon FreeRTOS OTA agent on the device receives the updated firmware image.
8. The device verifies the digital signature, checksum, and version number of the new image.
9. Reset the board and, based on application-defined logic, commit the update. This includes notifying AWS IoT Device Management that the device has (or has not) successfully completed the OTA update.

OTA Security

The following are three aspects of OTA security:

Connection security

The OTA Update service 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 mechanisims. 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 service uses the Code Signing for Amazon FreeRTOS service 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**
> The Code Signing for Amazon FreeRTOS feature is currently in beta. You can download the SDK from https://tools.signer.aws.a2z.com/awssigner-tools.zip.

Monitoring Updates

The code deployment manager `createDeployment` API returns the job ID from the AWS IoT Jobs service. You can use the job ID and the MQTT AWS IoT Jobs APIs to track the progress and status of the OTA updates across the fleet of the devices.

`DescribeJobExecution`

    Gets the details of a job execution. A job execution is an instance of a job running on a single device.

`ListJobExecutionsForJob`

    Lists all job executions for a job.

`ListJobExecutionsForThing`

    Gets the list of all job executions for a thing.

For more information, see AWS IoT Jobs API.

Amazon FreeRTOS Configuration Wizard User Guide

Managing Amazon FreeRTOS Configurations

You can use the AWS IoT Device Management 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 configurations are defined for the prequalified platforms:

- TI CC3220SF-LAUNCHXL
- STM32 IoT Discovery Kit
- NXP LPC54018 IoT Module
- Microchip Curiosity PIC32MZEF

- 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**.

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. On the **New Software Configuration** page, choose **Select a hardware platform**, and choose one of the prequalified platforms.
3. Choose the IDE and compiler you want use.
4. 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**.
5. In the **Demo Projects** section, enable one of the demo projects. This enables the demo in the project files.
6. In **Name required**, type a name for your custom configuration.

   Note
   Do not use any personally identifiable information for your custom configuration name.
7. In **Description**, you can type a description for your custom configuration.
8. At the bottom of the page, choose **Create and download** to create and download your custom configuration.

Deleting a Configuration

To delete an Amazon FreeRTOS configuration

1. Browse to the Amazon FreeRTOS console.
2. In the navigation pane, choose **Software**.
3. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
4. Choose the ellipsis next to the configuration you want to delete, and then choose **Delete**.

Downloading Amazon FreeRTOS from GitHub

Although we recommend that you download the Amazon FreeRTOS kernel and software libraries from the Amazon FreeRTOS console, all Amazon FreeRTOS files are available on GitHub.

Amazon FreeRTOS Qualification Program

This section provides information for MCU vendors about the Amazon FreeRTOS qualification workflow, which includes:

- Creating an Amazon FreeRTOS project.
- Porting Amazon FreeRTOS abstraction layers.

- Running tests.
- Submitting test results to the Amazon FreeRTOS team for final qualification.

What's in it for OEMs?

The Amazon FreeRTOS Qualification Program intends to give confidence to OEM/ODM developers that by using a qualified microcontroller (MCU) from this program for their IoT device, they can run Amazon FreeRTOS on the device without compatibility issues. It works with AWS IoT or AWS Greengrass. This allows OEM/ODM developers to focus on the code for their device functionality.

Qualification Program for MCU Vendors

The Amazon FreeRTOS Qualification Program gives MCU vendors confidence that their qualified MCUs are secure and interoperate with AWS IoT and AWS Greengrass. This means that the MCUs and associated libraries meet the security, functionality, and performance requirements to work seamlessly with AWS IoT and AWS Greengrass. A qualified MCU is included in the Amazon FreeRTOS console, where customers can select it and download its libraries. These include Amazon FreeRTOS and board support packages (BSPs) and drivers. Details of the qualified MCU, along with relevant links and the MCU vendor company logo, are available on the Amazon FreeRTOS Partners web page. The rest of this FAQ describes the steps to qualify your MCU and verify that your software (including drivers and BSPs) functionally integrates with Amazon FreeRTOS software.

Contact Amazon

If you want to qualify an MCU, send a request to <`freertos-qual@amazon.com`>. A representative from the qualification support team will scontact you and support you through the qualification steps.

Sign Up for the AWS Partner Network

The AWS Partner Network (APN) is the global partner program for AWS. It is focused on helping APN Partners build successful AWS-based businesses or solutions by providing business, technical, marketing, and go-to-market support. To find and register for the APN Partner program, see AWS Partner Network.

Jointly Agree on Terms and Conditions

After you become an APN Partner, you and AWS jointly review and agree on general terms and conditions, schedules, and initiatives in the partnership. The agreement includes topics such as the purpose of collaboration, alliance team, initiative development, marketing and collateral, indemnification, limitations of liability, and other general terms. After you and AWS sign the agreement, you can start the qualification workflow process.

Pass Qualification Test Suite

There are several steps to verify that your software (including drivers and BSPs) is functionally integrated with Amazon FreeRTOS software. The goal of this process is to create an MCU development project that successfully builds and runs a range of functional, performance, and security tests on your MCU.

The high-level steps are:

1. Download the latest version of the Amazon FreeRTOS source code.
2. Create a project using the target IDE and compiler that demonstrates the equivalent of a “Hello World” example for the target MCU.

3. Add Amazon FreeRTOS files and resources to the created project, and confirm the project still builds. At this stage, the included TQP tests should build and run, but are expected to fail because they have not yet been ported to your hardware.
4. Enable each Amazon FreeRTOS feature to work successfully on your MCU. This involves implementing each hardware-dependent layer of the Amazon FreeRTOS feature abstractions. Test these implementations and fix issues.
5. When all included tests pass, submit your results (test reports) and your microcontroller hardware to Amazon to confirm the qualification process.

Amazon FreeRTOS Qualified

After your hardware passes the verification tests, it is considered Amazon FreeRTOS-qualified. Information about your hardware will be displayed in the Amazon FreeRTOS console and the Amazon FreeRTOS Getting Started page.

Supported Platforms

Texas Instruments CC3220SF-LAUNCHXL

The SimpleLink Wi-Fi CC3220SF-LAUNCHXL LaunchPad Development Kit includes the CC3220SF, a single-chip wireless microcontroller (MCU) with ARM Cortex -M4 Core at 80 MHz, 1 MB Flash, 256 KB of RAM, and enhanced security features. The on-chip Wi-Fi module offloads TLS and TCP/IP processing, freeing up memory and compute for the application on the main microcontroller. For more information about this platform, see, Texas Instruments CC3220SF-LAUNCHXL.

STMicroelectronics STM32L4 Discovery Kit – IoT Node

The STM32L4 IoT Discovery Kit (B-L475E-IOT01A) supports Wi-Fi and integrates additional sensors. The kit has an STM32L4 Series MCU based on ARM Cortex -M4 core at 80 MHz with 1 MB of flash memory and 128 KB of SRAM, and Wi-Fi module Inventek ISM43362-M3G-L44. The Wi-Fi module offloads TCP/IP processing from the MCU. The interface to the Wi-Fi module for this kit has not yet been optimized for use with Amazon FreeRTOS, so there are limitations on its use. We recommend using the Secure Sockets APIs from low-priority tasks only, and to limit transmit throughput. Revisions to improve this interface are planned in the future. For more information about this platform, see STMicroelectronicsSTM32L4Discovery Kit IoT Node.

NXP LPC54108 IoT Module

The LPC54018 IoT Module from NXP includes an LPC54018 MCU with a 180MHz ARM Cortex-M4 core with 360 KB of SRAM, 128 MB of Quad-SPI Flash (Macronix MX25L12835FM2), and a Longsys IEEE802.11b/g/n Wi-Fi module based on Qualcomm GT1216. The Wi-Fi module offloads TCP/IP processing from the MCU. The LPC54018 IoT Module requires a debugger and J-Link connector (available in the NXP Direct store) or a baseboard. For more information about this platform, see NXP LPC54018 IoT Module.

Microchip Curiosity PIC32MZEF

The Curiosity PIC32MZEF development board from Microchip includes a PIC32MZEF MCU with a 200 MHz 32-bit MIPS M-class core with 2 MB of Flash and 512 KB of SRAM. For users who need to use Ethernet, the LAN8720A Ethernet PHY daughter board can be connected to the Curiosity PIC32MZEF development board. For more information about this platform, see Microchip PIC32MZ2048EFM100.

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

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:

```
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.

```
SOCKETS_SO_REQUIRE_TLS
SOCKETS_SO_SERVER_NAME_INDICATION
SOCKETS_SO_TRUSTED_SERVER_CERTIFICATE
```

For information about these nonstandard options, see the secure sockets documentation (p. 61). For information about porting TLS and cryptographic operations, see the TLS (p. 62) and Public Key Cryptography Standard #11 (p. 63) sections.

Error Codes

The SOCKETS library returns error codes from the API (rather than setting a global errno). All error codes returned must be negative values.

The public interfaces that must be ported are listed in `lib/secure_sockets/portable/`*`<vendor>`*`/`*`<platform>`*`/aws_secure_sockets.c`.

A detailed explanation for each public interface can be found in `lib/include/aws_secure_sockets.h`.

If you are using TLS based on mbed TLS, you can save refactoring effort by implementing network send and network receive functions that can be registered with the TLS layer for sending and receiving plaintext or encrypted buffers.

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 Greengrass devices:

- `TLS_RSA_WITH_AES_128_GCM_SHA256`

- `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_RSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA` (not recommended)
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA` (not recommended)
- `TLS_RSA_WITH_AES_128_CBC_SHA` (not recommended)
- `TLS_RSA_WITH_AES_256_CBC_SHA` (not recommended)

Due to attacks on SHA1, we recommend that you use SHA256 or SHA384 for Amazon FreeRTOS connections.

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.