Amazon FreeRTOS

User Guide

Amazon FreeRTOS: User Guide

Copyright © 2018 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What Is Amazon FreeRTOS? 1
- The FreeRTOS Kernel 1
- Amazon FreeRTOS Libraries 1
- Amazon FreeRTOS Configuration Wizard 1
- Over-the-Air Updates (Beta) 2
- Amazon FreeRTOS Qualification Program 2
- Development Workflow 2

Getting Started with Amazon FreeRTOS 4
- Prerequisites 4
  - AWS Account and Permissions 4
  - Amazon FreeRTOS Supported Hardware Platforms 5
  - Registering Your MCU Board with AWS IoT 5
  - Install a Terminal Emulator 7
- Getting Started with the Texas Instruments CC3220SF-LAUNCHXL 7
  - Setting Up Your Environment 7
  - Download and Build Amazon FreeRTOS 8
  - Run the FreeRTOS Samples 11
  - Troubleshooting 12
- Getting Started with the STMicroelectronics STM32L4 Discovery Kit IoT Node 13
  - Setting Up Your Environment 13
  - Download and Build Amazon FreeRTOS 13
  - Run the FreeRTOS Samples 16
- Getting Started with the NXP LPC54018 IoT Module 16
  - Setting Up Your Environment 16
  - Connecting a JTAG Debugger 17
  - Download and Build Amazon FreeRTOS 17
- Getting Started with the Microchip Curiosity PIC32MZEF 20
  - Setting Up the Microchip Curiosity PIC32MZEF Hardware 21
  - Setting Up Your Environment 22
  - Download and Build Amazon FreeRTOS 22
- Getting Started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT 26
  - Setting up the Espressif Hardware 26
  - Setting Up Your Environment 27
  - Download and Build Amazon FreeRTOS 27
  - Debugging Code on Espressif ESP32-DevKitC and ESP-WROVER-KIT 32
- Getting Started with the FreeRTOS Windows Simulator 36
  - Setting Up Your Environment 36
  - Download and Build Amazon FreeRTOS 36
  - Run the FreeRTOS Samples 39

Amazon FreeRTOS Developer Guide 40
- Amazon FreeRTOS Architecture 40
- FreeRTOS Kernel Fundamentals 41
  - FreeRTOS Kernel Scheduler 41
  - Memory Management 41
  - Inter-task Coordination 42
  - Software Timers 45
  - Low Power Support 45
- FreeRTOS Libraries 46
  - Cloud Connectivity 46
  - Greengrass Connectivity 48
  - Amazon FreeRTOS Security 50
  - FreeRTOS Wi-Fi Interface 54
  - OTA agent Library 55
- Amazon FreeRTOS Over the Air Updates 56

Over the Air Update Prerequisites ... 57
OTA Tutorial ... 70
OTA Update Manager Service ... 85
OTA Security ... 85
Setting Up Cloudwatch Logs for OTA Updates ... 86
Amazon FreeRTOS Configuration Wizard User Guide ... 90
Managing Amazon FreeRTOS Configurations ... 90
Downloading Amazon FreeRTOS from GitHub ... 91
Amazon FreeRTOS Qualification Program ... 91
What's in it for OEMs? ... 91
Qualification Program for MCU Vendors ... 91
Contact Amazon ... 92
Sign Up for the AWS Partner Network ... 92
Jointly Agree on Terms and Conditions ... 92
Pass Qualification Test Suite ... 92
Amazon FreeRTOS Qualified ... 92
Supported Platforms ... 92
Texas Instruments CC3220SF-LAUNCHXL ... 92
STMicroelectronics STM32L4 Discovery Kit – IoT Node ... 93
NXP LPC54108 IoT Module ... 93
Microchip Curiosity PIC32MZEF ... 93
Espressif ESP32-DevKitC ... 93
Espressif ESP32-WROVER-KIT ... 93
Amazon FreeRTOS Demo Projects ... 94
Navigating the Demo Applications ... 94
Directory and File Organization ... 94
Configuration Files ... 95
Device Shadow Demo Application ... 95
Greengrass Discovery Demo Application ... 97
OTA Demo Application ... 97
Amazon FreeRTOS Porting Guide ... 101
Bootloader ... 101
Logging ... 101
Logging Configuration ... 101
Connectivity ... 102
Wi-Fi Management ... 102
Sockets ... 102
Security ... 103
TLS ... 103
PKCS#11 ... 104
Using Custom Libraries with Amazon FreeRTOS ... 105
OTA Portable Abstraction Layer ... 105

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:

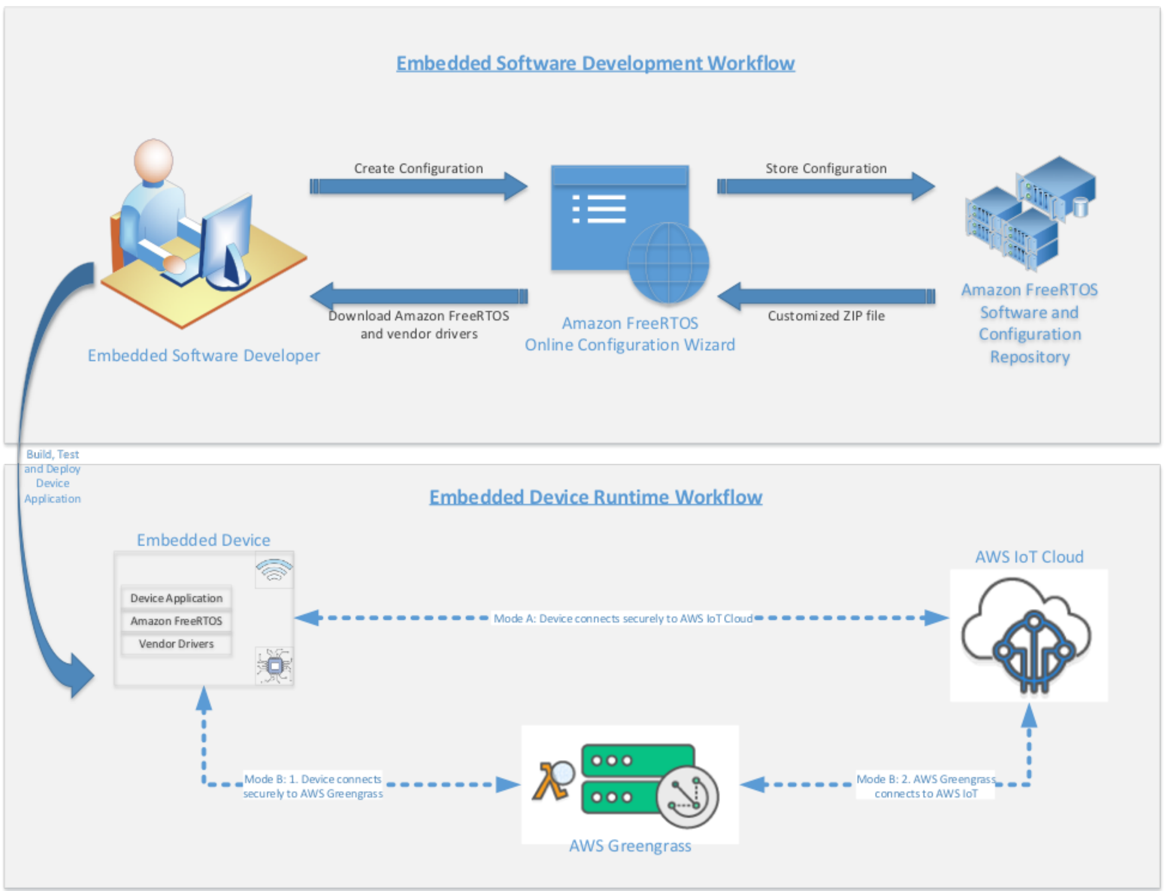
- Amazon FreeRTOS Over the Air Updates (p. 56)
- OTA Demo Application (p. 97)

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.



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
- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- 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 **Attach a policy** to attach a policy to your certificate that grants your device access to AWS IoT operations.
10. Choose the policy you just created and choose **Register thing**.

Install a Terminal Emulator

To verify your device code is running properly or to help in diagnosing problems, a terminal emulator can be very helpful. There are a variety of terminal emulators available for Windows, OS X, and Linux. You will need to connect your device to your computer before attempting to connect a terminal emulator to your device. Use these settings in your terminal emulator:

Terminal Setting	Value
Port	Depends on platform and what other devices you have connected to your computer.
BAUD rate	115200
Data	8 bit
Parity	none
Stop	1 bit
Flow control	none

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

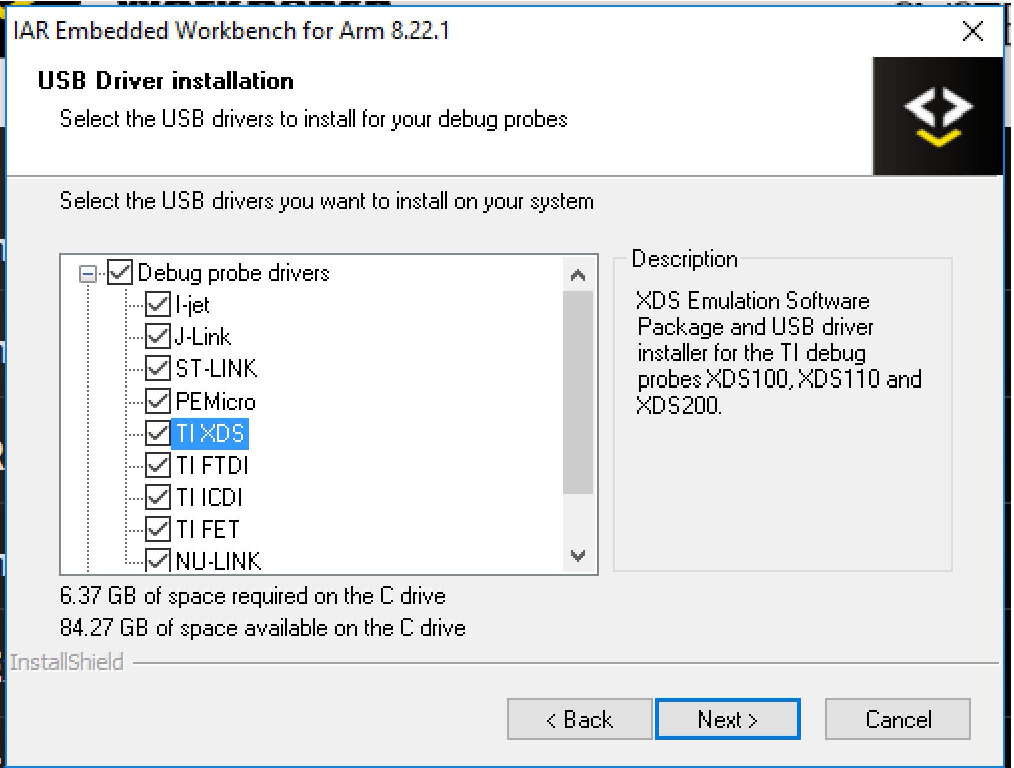
1. Browse to TI Code Composer Studio.
2. Download the offline installer for version 7.3.0 for the platform of your host machine (Windows, macOS, or Linux 64-bit).
3. Unzip and run the offline installer. Follow the prompts.
4. For **Product Families to Install**, choose **SimpleLink Wi-Fi CC32xx Wireless 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:



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

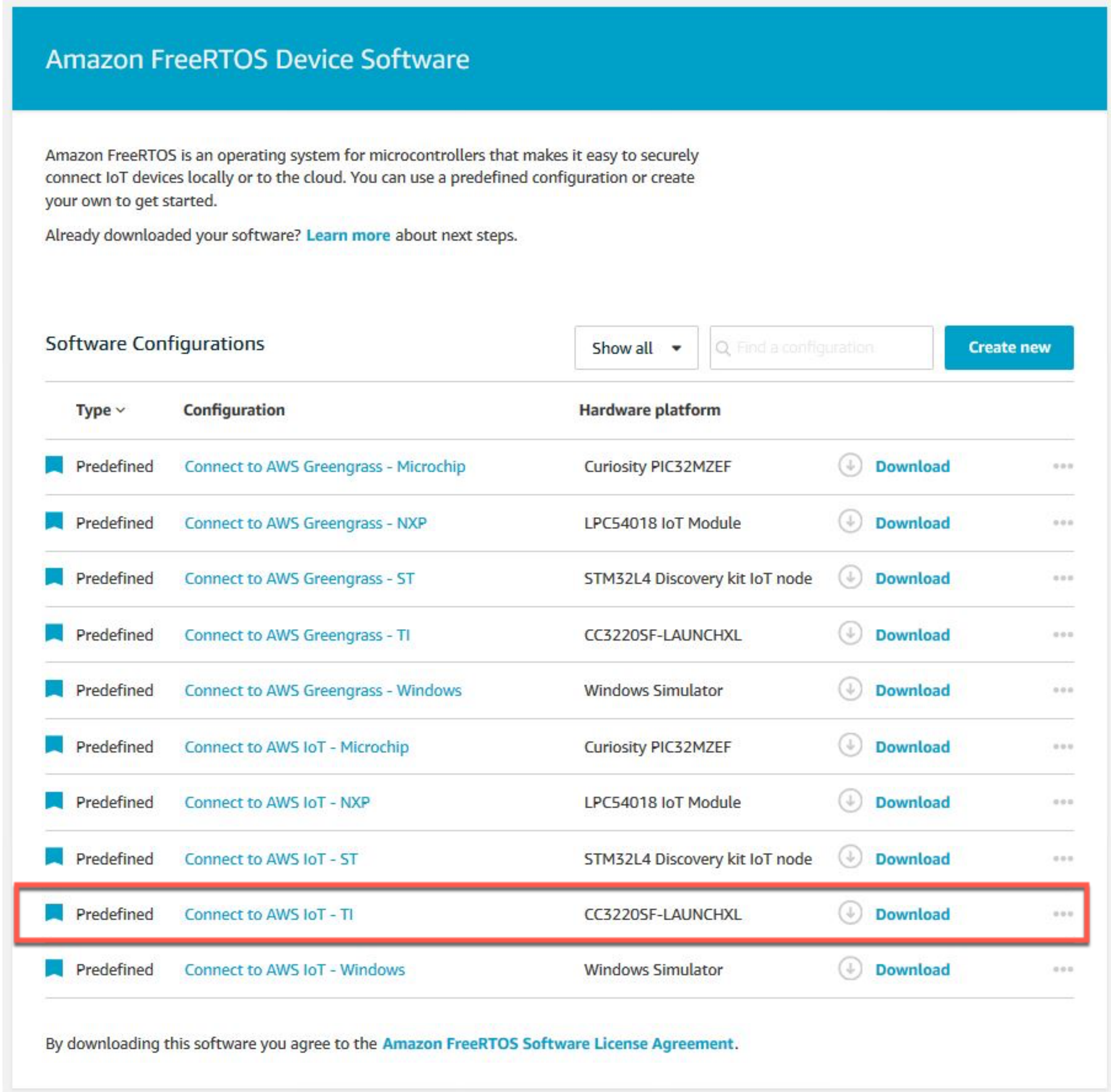
After your environment is set up, you can download Amazon FreeRTOS and run the sample code.

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**.



5. Unzip the downloaded file to your harddrive. When unzipped you will have a directory called AmazonFreeRTOS. You can place this directory anywhere you wish (see note below for path length limitations on Windows). In this tutorial, the path to the AmazonFreeRTOS directory 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. In the **Project Explorer** window, double click **aws_demos** to make the **aws_demos** project active.
5. 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

Open `aws_demos\application_code\common_demos\include\aws_clientcredential.h` in your IDE.

> **Note**
> The path listed above is what appears in the IDE's project or workspace window. The location of the file on your hard drive is `<BASE_FOLDER>\demos\common\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. This is for demo purposes only. Production level applications should store these files in a secure location. 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 (Sense On Power) jumper on your Texas Instruments CC3220SF-LAUNCHXL is in position 0. For more information, see CC3200 SimpleLink User's Guide.
2. Use a USB cable to connect your Texas Instruments CC3220SF-LAUNCHXL to your computer.
3. Sign in to the AWS IoT console.
4. In the 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**.

Run the FreeRTOS Samples in Code Composer Studio

1. Rebuild your project.
2. From the **Run** menu in TI Code Composer, choose **Debug** to start debugging.
3. 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.

Run the FreeRTOS Samples in IAR Embedded Workbench

1. Rebuild the project by going to the **Project** menu and choosing **Make**.

2. In the **Project** menu select **Download and Debug**. If you see dialog stating "Warning: Failed to initialize EnergyTrace" you can close the dialog and ignore the warning. For more information about EnergyTrace, see MSP EnergyTrace Technology.
3. When the debugger stops at the breakpoint in `main()`, go to the **Debug** menu, and then choose **Go**.

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.

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

Setting Up Your Environment

Install System Workbench for STM32

1. Browse to OpenSTM32.org.
2. Register on the OpenSTM32 webpage. You need to sign in to download System Workbench.
3. Browse to the System Workbench for STM32 installer to download and install System Workbench.

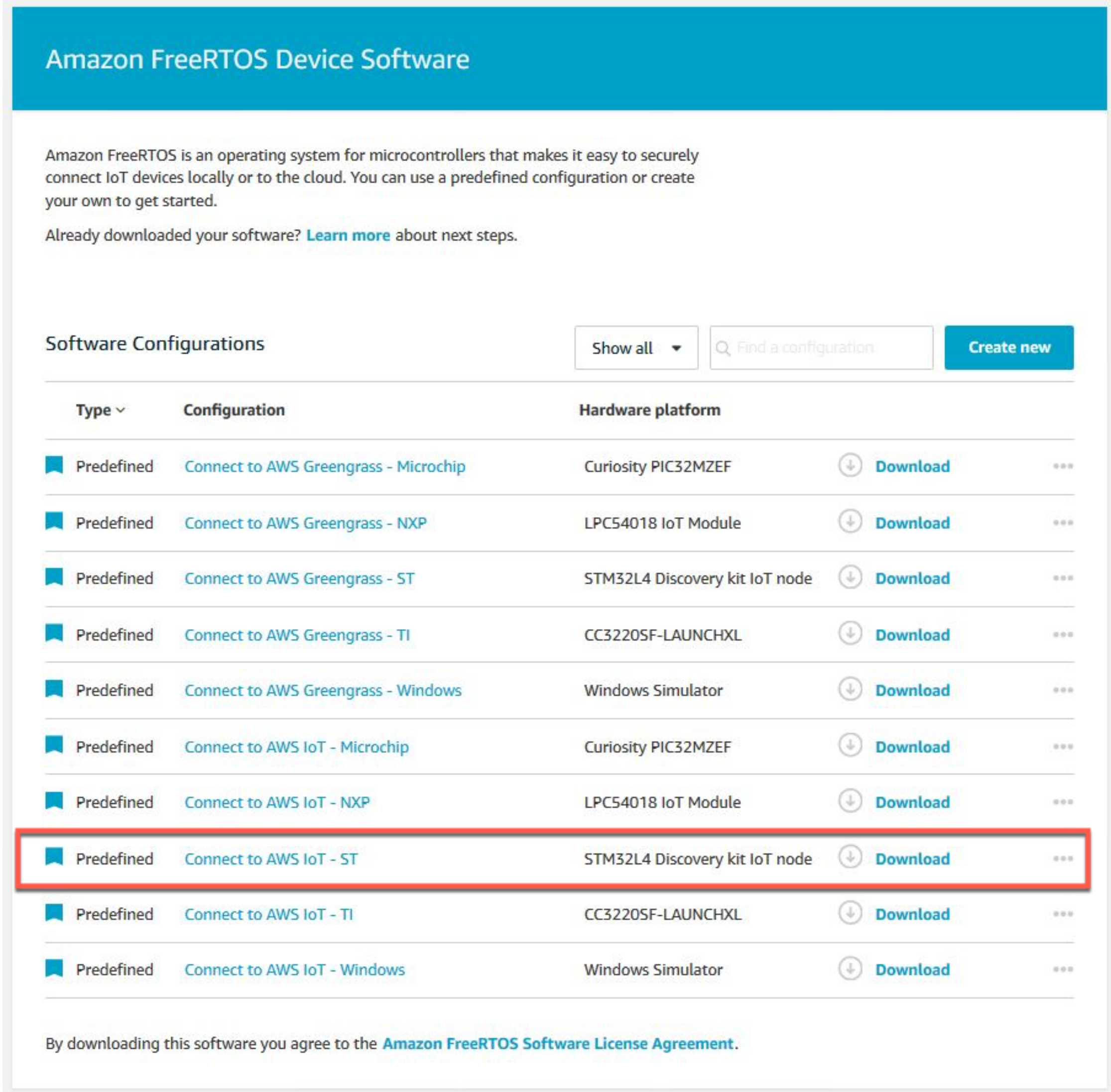
If you experience issues during installation, see the FAQs on the System Workbench website.

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**.



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. This is for demo purposes only. Production level applications should store these files in a secure location. 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. This is for demo purposes only. Production level applications should store these files in a secure location. 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).

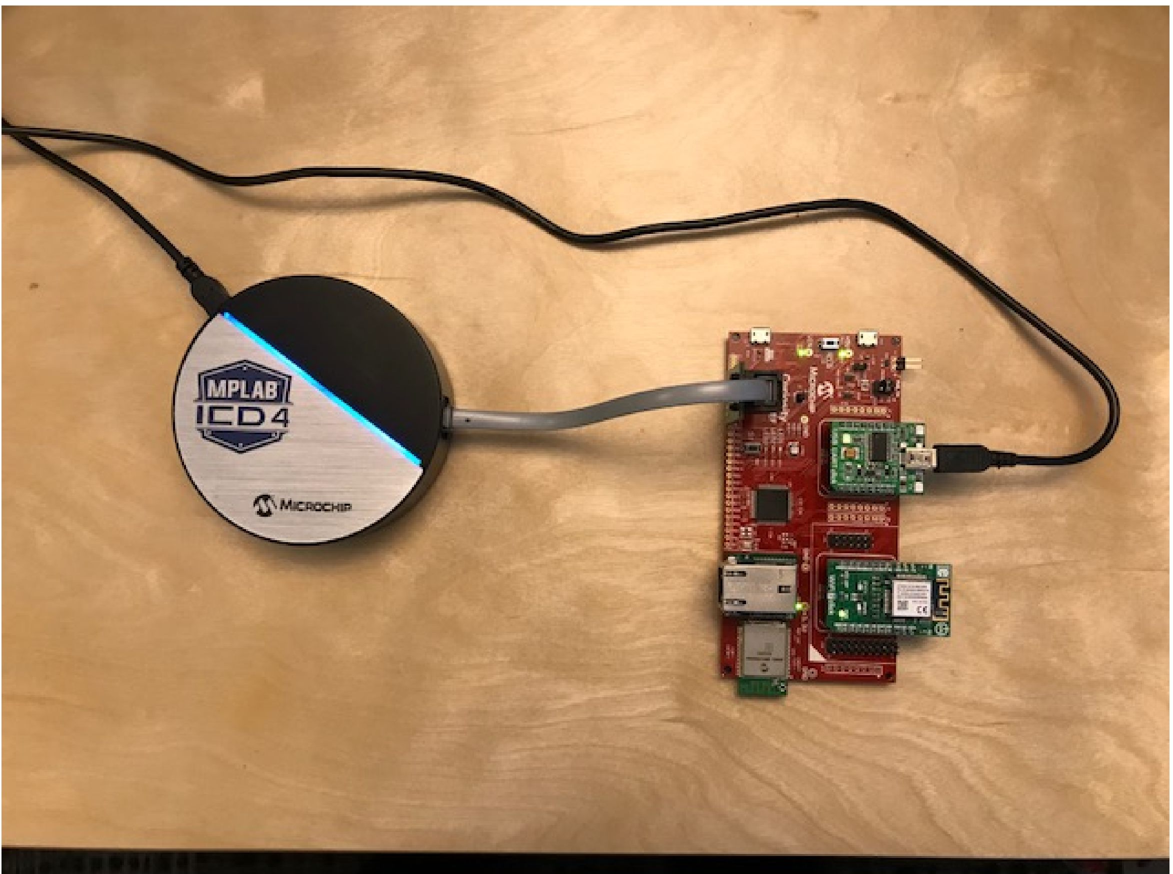
If you do not have the Microchip Curiosity PIC32MZEF bundle, you can purchase one from Microchip. You need the following items:

- 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

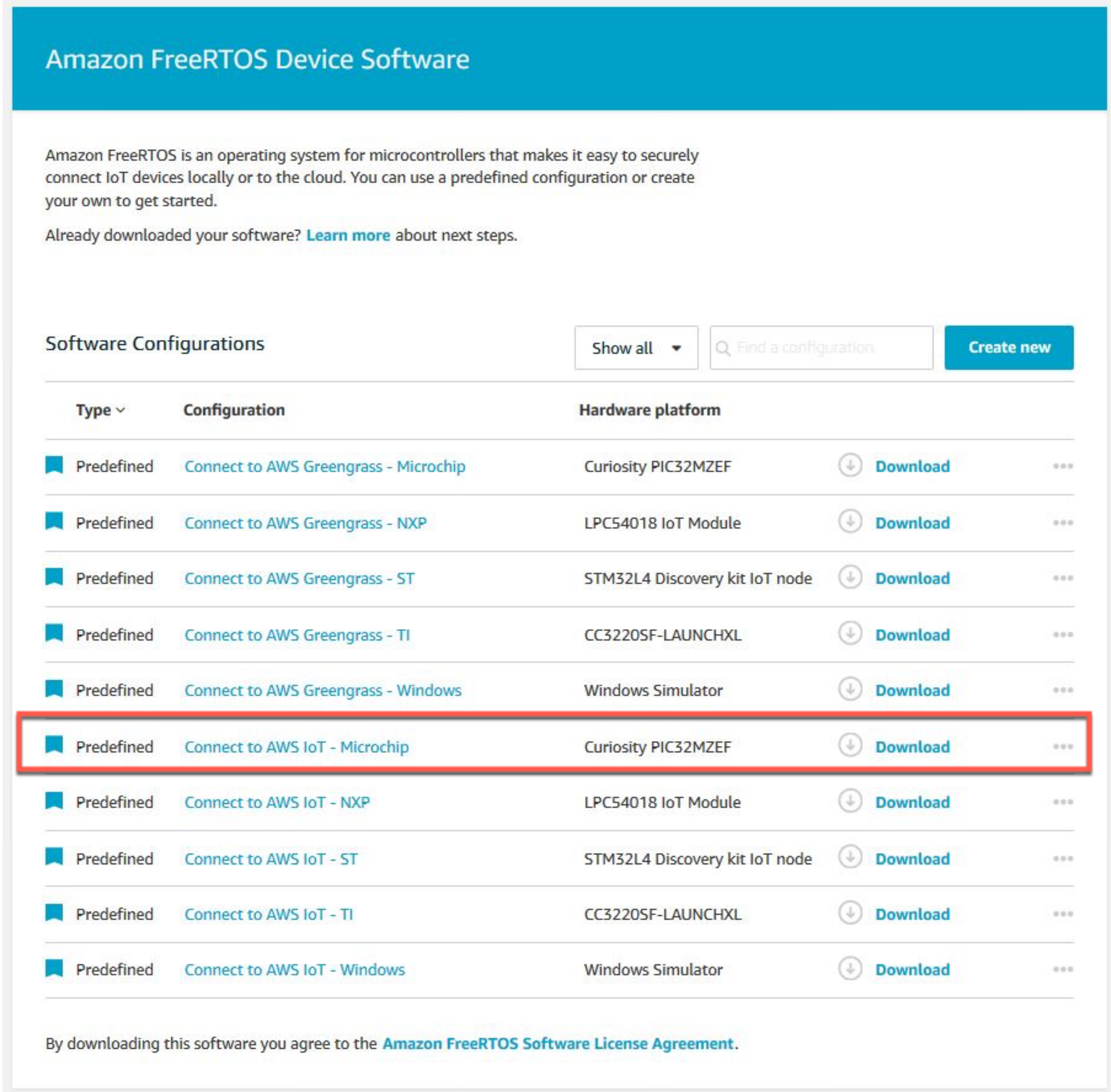
1. Install the latest Java SE SDK.
2. Install Python version 3.x or greater.
3. Install the latest version of the MPLAB X Integrated Development Environment:
   - MPLAB X Integrated Development Environment for Windows
   - MPLAB X Integrated Development Environment for macOS
   - MPLAB X Integrated Development Environment for Linux
4. Install the latest version of the MPLAB XC32 Compiler:
   - MPLAB XC32/32++ Compiler for Windows
   - MPLAB XC32/32++ Compiler for macOS
   - MPLAB XC32/32++ Compiler for Linux
5. Install the latest version of the MPLAB Harmony Integrated Software Framework (optional):
   - MPLAB Harmony Integrated Software Framework for Windows
   - MPLAB Harmony Integrated Software Framework for macOS
   - MPLAB Harmony Integrated Software Framework for Linux
6. Start up a UART terminal emulator and open a connection with the following settings:
   - Baud rate: 115200
   - Data: 8 bit
   - Parity: None
   - Stop bits: 1
   - Flow control: None

Download and 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**.



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. This is for demo purposes only. Production level applications should store these files in a secure location. 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. 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. On the **Projects** tab, right-click the aws_demos top-level folder and then choose **Debug**.
6. The first time you debug the samples, an **ICD 4 not Found** dialog box appears. In the tree view, under the **ICD 4** node, choose the ICD4 serial number, and then choose **OK**.
7. When the debugger stops at the breakpoint in `main()`, go to the **Run** menu, and then choose **Resume**.

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

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


Connecting to MPLAB ICD 4...

Currently loaded versions:
Application version............01.02.00
Boot version...................01.00.00
FPGA version...................01.00.00
Script version.................00.02.18
Script build number............fd44437f19
Application build number.......0123456789

Connecting to MPLAB ICD 4...

Currently loaded versions:
Boot version...................01.00.00
Updating firmware application...
Connecting to MPLAB ICD 4...

Currently loaded versions:
Application version............01.02.16
Boot version...................01.00.00
FPGA version...................01.00.00
Script version.................00.02.18
Script build number............fd44437f19
Application build number.......0123456789

Target voltage detected
```

```
Target device PIC32MZ2048EFM100 found.
Device Id Revision = 0xA1
Serial Number:
Num0 = ec4f6d3c
Num1 = 6b845410

Erasing...

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

Programming/Verify complete

Running
```

Note

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

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 Espressif ESP32-DevKitC and the ESP-WROVER-KIT

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

Note

Amazon FreeRTOS's port for ESP32-WROVER-KIT and ESP DevKitC currently does not support the following:

1. Over the Air (OTA) update functionality.
2. Lightweight IP.
3. Support for symmetric multiprocessing (SMP).
4. Bluetooth Low Energy (BLE).

Setting up the Espressif Hardware

For the ESP32-DevKitC development board, please refer to the Getting Started with the ESP32-DevKitC development board.

For the ESP-WROVER-KIT development board, please refer to the Getting Started with the ESP-WROVER-KIT development board.

Setting Up Your Environment

Establishing a Serial Connection

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

For more information see, Establishing a Serial Connection with ESP32.

Note the serial port you configure (based on host OS), you will need it during the build process.

Setting Up the Toolchain

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

- Setting up the toolchain for Windows
- Setting up the toolchain for Mac OS
- Setting up the toolchain for Linux

Download and Build Amazon FreeRTOS

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

Downloading Amazon FreeRTOS

Clone the Amazon FreeRTOS repository from GitHub. This document assumes you have cloned the repository into a directory called *`BASE_FOLDER`*.

> **Note**
> The maximum length of a file path on Microsoft Windows is 260 characters. The longest path in the Amazon FreeRTOS repository 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.

Configure Your Project

1. If you are running MacOS or Linux, open a terminal prompt. If you are running Windows, open mingw32.exe.
2. Install python on your system. Minimum version should be 2.7.10.
3. If you are running on Windows, install the AWS CLI inside the mingw32 enviroment using the following command: `easy_install awscli`. If you are running on MacOS or Linux, make sure the AWS CLI is installed on your system. For more information about installing the AWS CLI, see Installing the AWS Command Line Interface.
4. Configure the AWS CLI by running `aws configure`. For information about configuring the AWS CLI, see Configuring the AWS CLI.
5. Install the boto3 Python module using the following command:

   - On Windows in the mingw32 environment: `easy_install boto3`

- On MacOS or Linux: `pip install boto3`

Amazon FreeRTOS comes with a set of scripts that makes setting up your Espressif board easier. To configure the Espressif scripts, open *<BASE_FOLDER>*/demos/common/tools/aws_config_quick_start/configure.json and set the following attributes:

`thing_name`

The name of the IoT thing that represents your board.

`wifi_ssid`

The SSID of your Wi-Fi network.

`wifi_password`

The password for your Wi-Fi network.

`wifi_security`

The security type for your Wi-Fi network. Valid security types are:

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

To run the configuration script:

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

   ```
   python SetupAWS.py setup
   ```

This script will create an IoT thing, certificate, and an IoT policy. It will attach the IoT policy to the certificate and the certificate to the IoT thing. It will also populate the file aws_clientcredential.h with your AWS IoT endpoint, Wi-Fi SSID and credentials. Finally it will format your certificate and private key and write them to the aws_clientcredential_keys.h header file. For further information about the script, please refer the README.md in the *<BASE_FOLDER>*`/demos/common/tools/aws_config_quick_start` directory.

Run the FreeRTOS Samples

To flash the demo application onto your board

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

   ```
   make menuconfig
   ```

An Espressif IoT Development Framework Configuration menu will be displayed.

In the menu, navigate to **Serial flasher config** and then **Default serial port** to configure the serial port.

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

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

Optionally, you can increase the default baud rate up to 921600 (depending upon your hardware), by choosing **Serial flash config** and then choosing **Default baud rate**. This can reduce the time required to flash your board.

Confirm your selection by pressing [ENTER], save the configuration by choosing **< Save >** and then exit application by choosing **< Exit >**.

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

```
make flash monitor
```

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

```

I (31) boot: ESP-IDF v3.1-dev-322-gf307f41-dirty 2nd stage bootloader
I (31) boot: compile time 11:30:50
I (34) boot: Enabling RNG early entropy source...
I (37) boot: SPI Speed      : 40MHz
I (41) boot: SPI Mode       : DIO
I (45) boot: SPI Flash Size : 4MB
I (49) boot: Partition Table:
I (53) boot: ## Label            Usage          Type ST Offset   Length
I (60) boot:  0 nvs              WiFi data        01 02 00009000 00006000
I (68) boot:  1 phy_init         RF data          01 01 0000f000 00001000
I (75) boot:  2 factory          factory app      00 00 00010000 00100000
I (82) boot:  3 storage          Unknown data     01 82 00110000 00010000
I (90) boot: End of partition table
I (94) esp_image: segment 0: paddr=0x00010020 vaddr=0x3f400020 size=0x12710 ( 75536) map
I (129) esp_image: segment 1: paddr=0x00022738 vaddr=0x3ffb0000 size=0x0240c (  9228) load
I (133) esp_image: segment 2: paddr=0x00024b4c vaddr=0x40080000 size=0x00400 (  1024) load
0x40080000: _iram_start at /Users/michgre/Desktop/AmazonFreeRTOS-Espressif/lib/FreeRTOS/
portable/GCC/Xtensa_ESP32/xtensa_vectors.S:1685

I (136) esp_image: segment 3: paddr=0x00024f54 vaddr=0x40080400 size=0x0b0bc ( 45244) load
I (164) esp_image: segment 4: paddr=0x00030018 vaddr=0x400d0018 size=0x6d454 (447572) map
0x400d0018: _stext at ??:?

I (319) esp_image: segment 5: paddr=0x0009d474 vaddr=0x4008b4bc size=0x02d44 ( 11588) load
0x4008b4bc: xStreamBufferSend at /Users/michgre/Desktop/AmazonFreeRTOS-Espressif/lib/
FreeRTOS/stream_buffer.c:636

I (324) esp_image: segment 6: paddr=0x000a01c0 vaddr=0x400c0000 size=0x00000 (     0) load
I (334) boot: Loaded app from partition at offset 0x10000
I (334) boot: Disabling RNG early entropy source...
I (338) cpu_start: Pro cpu up.
I (341) cpu_start: Single core mode
I (346) heap_init: Initializing. RAM available for dynamic allocation:
I (353) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
```

```
I (359) heap_init: At 3FFC0420 len 0001FBE0 (126 KiB): DRAM
I (365) heap_init: At 3FFE0440 len 00003BC0 (14 KiB): D/IRAM
I (371) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (378) heap_init: At 4008E200 len 00011E00 (71 KiB): IRAM
I (384) cpu_start: Pro cpu start user code
I (66) cpu_start: Starting scheduler on PRO CPU.
I (96) wifi: wifi firmware version: f79168c
I (96) wifi: config NVS flash: enabled
I (96) wifi: config nano formating: disabled
I (106) system_api: Base MAC address is not set, read default base MAC address from BLK0 of
 EFUSE
I (106) system_api: Base MAC address is not set, read default base MAC address from BLK0 of
 EFUSE
I (136) wifi: Init dynamic tx buffer num: 32
I (136) wifi: Init data frame dynamic rx buffer num: 32
I (136) wifi: Init management frame dynamic rx buffer num: 32
I (136) wifi: wifi driver task: 3ffc5ec4, prio:23, stack:4096
I (146) wifi: Init static rx buffer num: 10
I (146) wifi: Init dynamic rx buffer num: 32
I (156) wifi: wifi power manager task: 0x3ffcc248 prio: 21 stack: 2560
0 7 [Tmr Svc] WiFi module initialized. Connecting to AP Guest...
W (166) phy_init: failed to load RF calibration data (0x1102), falling back to full
 calibration
I (396) phy: phy_version: 383.0, 79a622c, Jan 30 2018, 15:38:06, 0, 2
I (406) wifi: mode : sta (30:ae:a4:4b:3d:64)
I (406) WIFI: SYSTEM_EVENT_STA_START
I (526) wifi: n:1 0, o:1 0, ap:255 255, sta:1 0, prof:1
I (526) wifi: state: init -> auth (b0)
I (536) wifi: state: auth -> assoc (0)
I (536) wifi: state: assoc -> run (10)
I (536) wifi: connected with Guest, channel 1
I (536) WIFI: SYSTEM_EVENT_STA_CONNECTED
I (3536) wifi: pm start, type:0

1 826 [IP-task] vDHCPProcess: offer c0a8520bip
I (8356) event: sta ip: 192.168.82.11, mask: 255.255.224.0, gw: 192.168.64.1
I (8356) WIFI: SYSTEM_EVENT_STA_GOT_IP
2 827 [IP-task] vDHCPProcess: offer c0a8520bip
3 828 [Tmr Svc] WiFi Connected to AP. Creating tasks which use network...
4 828 [Tmr Svc] Creating MQTT Echo Task...
5 829 [MQTTEcho] MQTT echo attempting to connect to a14o5vz6c0ikzv.iot.us-
west-2.amazonaws.com.
6 829 [MQTTEcho] Sending command to MQTT task.
7 830 [MQTT] Received message 10000 from queue.
8 2030 [IP-task] Socket sending wakeup to MQTT task.
I (20416) PKCS11: Initializing SPIFFS
W (20416) SPIFFS: mount failed, -10025. formatting...
I (20956) PKCS11: Partition size: total: 52961, used: 0
9 2596 [MQTT] Received message 0 from queue.
10 2601 [IP-task] Socket sending wakeup to MQTT task.
11 2601 [MQTT] Received message 0 from queue.
12 2607 [IP-task] Socket sending wakeup to MQTT task.
13 2607 [MQTT] Received message 0 from queue.
14 2607 [MQTT] MQTT Connect was accepted. Connection established.
15 2607 [MQTT] Notifying task.
16 2608 [MQTTEcho] Command sent to MQTT task passed.
17 2608 [MQTTEcho] MQTT echo connected.
18 2608 [MQTTEcho] MQTT echo test echoing task created.
19 2608 [MQTTEcho] Sending command to MQTT task.
20 2609 [MQTT] Received message 20000 from queue.
21 2610 [IP-task] Socket sending wakeup to MQTT task.
22 2611 [MQTT] Received message 0 from queue.
23 2612 [IP-task] Socket sending wakeup to MQTT task.
24 2612 [MQTT] Received message 0 from queue.
25 2612 [MQTT] MQTT Subscribe was accepted. Subscribed.
26 2612 [MQTT] Notifying task.
```

```
27 2613 [MQTTEcho] Command sent to MQTT task passed.
28 2613 [MQTTEcho] MQTT Echo demo subscribed to freertos/demos/echo
29 2613 [MQTTEcho] Sending command to MQTT task.
30 2614 [MQTT] Received message 30000 from queue.
31 2619 [IP-task] Socket sending wakeup to MQTT task.
32 2619 [MQTT] Received message 0 from queue.
33 2620 [IP-task] Socket sending wakeup to MQTT task.
34 2620 [MQTT] Received message 0 from queue.
35 2620 [MQTT] MQTT Publish was successful.
36 2620 [MQTT] Notifying task.
37 2620 [MQTTEcho] Command sent to MQTT task passed.
38 2620 [MQTTEcho] Echo successfully published 'Hello World 0'
39 2624 [IP-task] Socket sending wakeup to MQTT task.
40 2624 [MQTT] Received message 0 from queue.
41 2624 [Echoing] Sending command to MQTT task.
42 2624 [MQTT] Received message 40000 from queue.
43 2625 [IP-task] Socket sending wakeup to MQTT task.
44 2625 [MQTT] Received message 0 from queue.
45 2626 [IP-task] Socket sending wakeup to MQTT task.
46 2626 [MQTT] Received message 0 from queue.
47 2628 [IP-task] Socket sending wakeup to MQTT task.
48 2628 [MQTT] Received message 0 from queue.
49 2628 [MQTT] MQTT Publish was successful.
50 2628 [MQTT] Notifying task.
51 2628 [Echoing] Command sent to MQTT task passed.
52 2630 [Echoing] Message returned with ACK: 'Hello World 0 ACK'

*** Similar output deleted for brevity ***

317 7692 [IP-task] Socket sending wakeup to MQTT task.
318 7692 [MQTT] Received message 0 from queue.
319 7698 [IP-task] Socket sending wakeup to MQTT task.
320 7698 [MQTT] Received message 0 from queue.
321 8162 [MQTTEcho] Sending command to MQTT task.
322 8162 [MQTT] Received message 190000 from queue.
323 8163 [IP-task] Socket sending wakeup to MQTT task.
324 8163 [MQTT] Received message 0 from queue.
325 8164 [IP-task] Socket sending wakeup to MQTT task.
326 8164 [MQTT] Received message 0 from queue.
327 8164 [MQTT] MQTT Publish was successful.
328 8164 [MQTT] Notifying task.
329 8165 [MQTTEcho] Command sent to MQTT task passed.
330 8165 [MQTTEcho] Echo successfully published 'Hello World 11'
331 8167 [IP-task] Socket sending wakeup to MQTT task.
332 8167 [MQTT] Received message 0 from queue.
333 8168 [Echoing] Sending command to MQTT task.
334 8169 [MQTT] Received message 1a0000 from queue.
335 8170 [IP-task] Socket sending wakeup to MQTT task.
336 8170 [MQTT] Received message 0 from queue.
337 8171 [IP-task] Socket sending wakeup to MQTT task.
338 8171 [MQTT] Received message 0 from queue.
339 8171 [MQTT] MQTT Publish was successful.
340 8171 [MQTT] Notifying task.
341 8172 [Echoing] Command sent to MQTT task passed.
342 8173 [Echoing] Message returned with ACK: 'Hello World 11 ACK'
343 8174 [IP-task] Socket sending wakeup to MQTT task.
344 8174 [MQTT] Received message 0 from queue.
345 8179 [IP-task] Socket sending wakeup to MQTT task.
346 8179 [MQTT] Received message 0 from queue.
347 8665 [MQTTEcho] Sending command to MQTT task.
348 8665 [MQTT] Received message 1b0000 from queue.
349 8665 [MQTT] About to close socket.
350 8666 [IP-task] Socket sending wakeup to MQTT task.
351 8667 [MQTT] Socket closed.
352 8667 [MQTT] Stack high watermark for MQTT task: 2792
353 8667 [MQTT] Notifying task.
```

```
354 8667 [MQTT] Received message 0 from queue.
355 8668 [MQTTEcho] Command sent to MQTT task passed.
356 8668 [MQTTEcho] MQTT echo demo finished.
```

Troubleshooting

- If you are using a Mac and it does not recognize your ESP-WROVER-KIT make sure you do not have the D2XX drivers installed. You can uninstall them following the instructions in the FTDI Drivers Installation Guide for Mac OSX.
- The ESP-IDF provided monitor utility (invoked using make monitor) helps in decoding addresses, thus helps in getting some meaningful backtraces in case the application crashes. For more information, see Automatically decoding addresses.
- It is also possible to enable GDBstub for communication with gdb without requiring any special JTAG hardware, for more information, see Launch GDB for GDBStub.
- If full-fledged JTAG hardware-based debugging is required, see JTAG Debugging for information about setting up an OpenOCD based environment.
- If `pyserial` cannot be installed using `pip` on MacOS, download it from pyserial.
- If the board resets continuously, try erasing the flash by typing the following command on the terminal:

```
make erase_flash
```

- If you see errors when running `idf_monitor.py`, please use Python 2.7.

Additional Notes

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

```
make flash monitor -j4
```

Debugging Code on Espressif ESP32-DevKitC and ESP-WROVER-KIT

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

ESP-DevKitC JTAG Setup

For the FTDI C232HM-DDHSL-0 cable the connections to the ESP32 DevkitC are as follows:

C232HM-DDHSL-0 Wire Color	ESP32 GPIO Pin	JTAG Signal Name
Brown (pin 5)	IO14	TMS
Yellow (pin 3)	IO12	TDI
Black (pin 10)	GND	GND

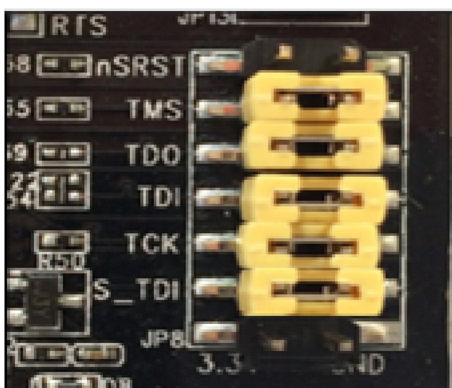
C232HM-DDHSL-0 Wire Color	ESP32 GPIO Pin	JTAG Signal Name
Orange (pin 2)	IO13	TCK
Green (pin 4)	IO15	TDO

ESP-WROVER-KIT JTAG Setup

For the FTDI C232HM-DDHSL-0 cable the connections to the ESP32-WROVER-KIT are as follows:

C232HM-DDHSL-0 Wire Color	ESP32 GPIO Pin	JTAG Signal Name
Brown (pin 5)	IO14	TMS
Yellow (pin 3)	IO12	TDI
Orange (pin 2)	IO13	TCK
Green (pin 4)	IO15	TDO

To enable JTAG on the ESP-WROVER-KIT, place jumpers on the TMS, TDO. TDI, TCK, and S_TDI pins as shown below:



The tables above were developed from the FTDI C232HM-DDHSL-0 datasheet, see section C232HM MPSSE Cable connection and Mechanical Details in the data sheet for more information.

Debugging on Windows

To set up for debugging on Windows

1. Connect the USB side of the FTDI C232HM-DDHSL-0 to your computer and the other side as described in the previous section. The FTDI C232HM-DDHSL-0 device should be listed in the **Device Manager** under **Universal Serial Bus Controllers**.
2. From the list of USB controllers, right-click the FTDI C232HM-DDHSL-0 device (the manufacturer is FTDI) and choose **Properties**. In the properties window, select the **Details** tab to see the properties of the device. If this device is not listed, install the Windows driver for FTDI C232HM-DDHSL-0.
3. Verify the vendor ID and product ID you saw in **Device Manager** matches the IDs in `demos\espressif\esp32_devkitc_esp_wrover_kit\esp32_devkitj_v1.cfg`. The ID are specified on a line that begins with `ftdi_vid_pid` followed by a vendor ID and a product ID, for example:

```
ftdi_vid_pid 0x0403 0x6014
```

4. Download OpenOCD for Windows.
5. Unzip the file to `C:\` and add `C:\openocd-esp32\bin` to your system path.
6. OpenOCD requires libusb which is not installed by default on Windows. To install it:

   a. Download zadig.exe.
   b. Run `zadig.exe`, from the **Options** menu select **List All Devices**.
   c. In the dropdown menu, choose **C232HM-DDHSL-0**.
   d. In the target driver listbox, to the right of the green arrow, choose **WinUSB**.
   e. In the dropdown box below the target driver listbox, click the arrow and select **Install Driver**. The control should then display **Replace Driver**. Click **Replace Driver**.
7. Open a command prompt, navigate to *<BASE_FOLDER>\demos\espressif\esp32_devkitc_esp_wrover_kit\make* and run:

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

   Leave this command prompt open.
8. Open a new command prompt, navigate to your `msys32` directory and run `mingw32.exe`. In the mingw32 terminal navigate to *<BASE_FOLDER>*`\demos\espressif\esp32_devkitc_esp_wrover_kit\make` and run `make flash monitor`.
9. Open another mingw32 terminal, navigate to *<BASE_FOLDER>*`\demos\espressif\esp32_devkitc_esp_wrover_kit\make` and run `xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf`. The program should stop in the `main` function.

Note
The ESP32 supports a maximum of two break points.

Debugging on Mac OS

1. Download the FTDI driver for MacOS.
2. Download OpenOCD for MacOS.
3. Extract the downloaded tar file and set the path in `.bash_profile` to *<OCD_INSTALL_DIR>*`/openocd-esp32/bin`.
4. Install `libusb` on MacOS with following command:

```
brew install libusb
```

5. Unload the serial port driver using the following command:

```
sudo kextunload -b com.FTDI.driver.FTDIUSBSerialDriver
```

6. If you are running MacOS version greater than 10.9, unload Apple's FTDI driver by using the following command:

```
sudo kextunload -b com.apple.driver.AppleUSBFTDI
```

7. Get the product ID and vendor ID of FTDI cable by listing the attached USB devices using the following command:

```
system_profiler SPUSBDataType
```

   The output from `system_profile` should look like the following:

```
C232HM-DDHSL-0:
Product ID: 0x6014
Vendor ID: 0x0403 (Future Technology Devices International Limited)
```

8. Verify the vendor ID and product ID matches the IDs in `demos/espressif/esp32_devkitc_esp_wrover_kit/esp32_devkitj_v1.cfg`. The ID are specified on a line that begins with `ftdi_vid_pid` followed by a vendor ID and a product ID, for example:

```
ftdi_vid_pid 0x0403 0x6014
```

9. Open a terminal window, navigate to *<BASE_FOLDER>*`/demos/espressif/esp32_devkitc_esp_wrover_kit/make`, and run OpenOCD with the following command:

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

10. Open a new terminal, load the FTDI serial port driver using the following command:

```
sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
```

11. Navigate to *<BASE_FOLDER>*`>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` and run:

```
make flash monitor
```

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

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

    The program should stop at `main()`.

Debugging on Linux

1. Download the FTDI driver for Linux.
2. Download OpenOCD for Linux. Extract the tarball and follow the installation instructions in the readme file.
3. Install libusb on Linux with following command:

```
sudo apt-get install libusb-1.0
```

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

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

5. Log off and log in and then cycle the power to the board to make the changes effective. In a terminal prompt list the USB devices and make sure the group-owner has changed from `dialout` to `plugdev`:

```
$ls -l /dev/ttyUSB*
```

```
crw-rw---- 1 root plugdev 188, 0 Jul 10 19:04 /dev/ttyUSB0
crw-rw---- 1 root plugdev 188, 1 Jul 10 19:04 /dev/ttyUSB1
```

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

6. In a terminal window, navigate to *<BASE_FOLDER>*`/demos/espressif/esp32_devkitc_esp_wrover_kit/make` and run OpenOCD:

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

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

```
make flash monitor
```

8. Open another terminal, navigate to *<BASE_FOLDER>*`/demos/espressif/esp32_devkitc_esp_wrover_kit/make` and run:

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

The program should stop in `main()`.

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.

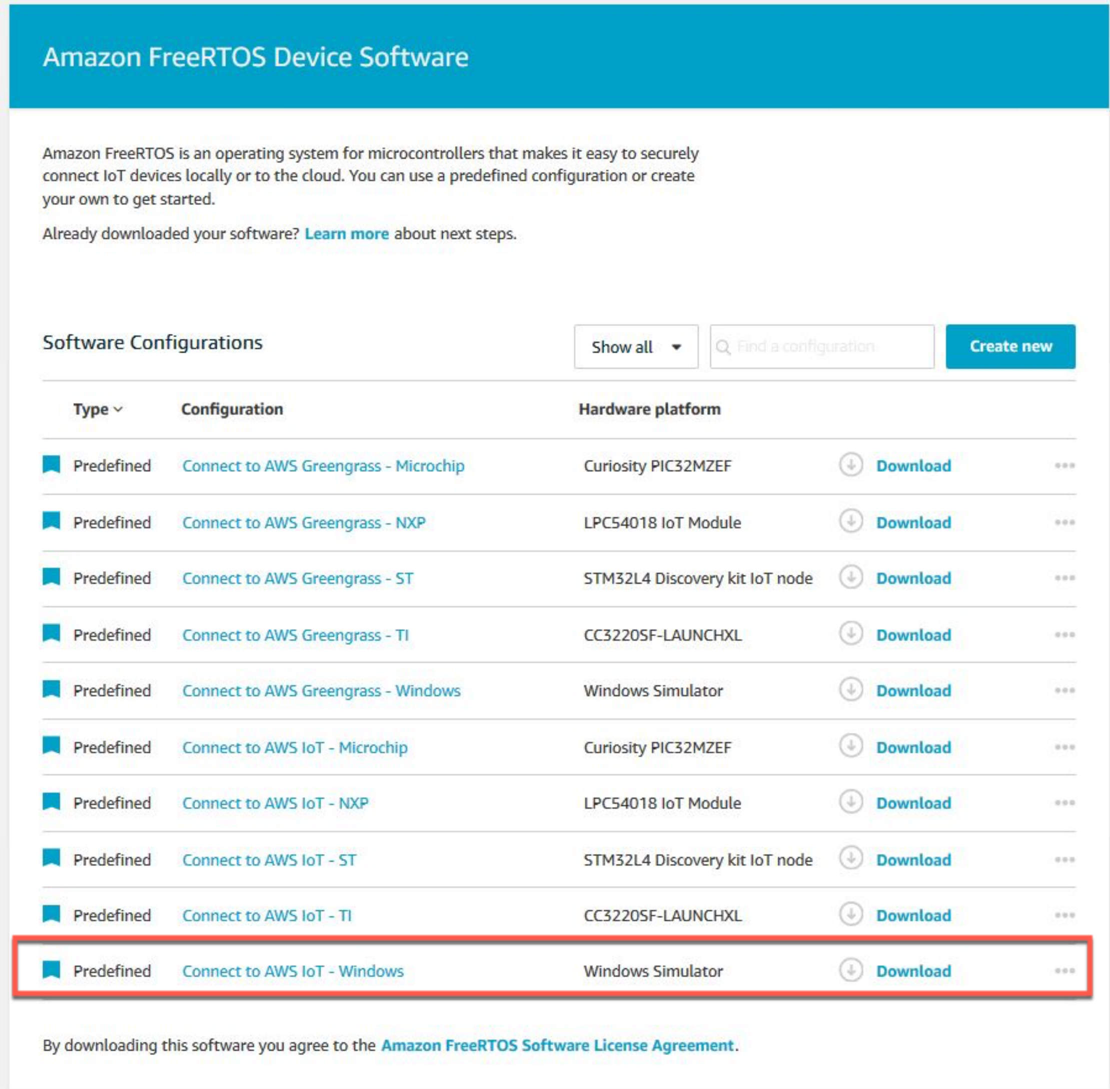
Download and Build Amazon FreeRTOS

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**.



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 demo code. This is for demo purposes only. Production level applications should store these files in a secure location. 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 Developer Guide

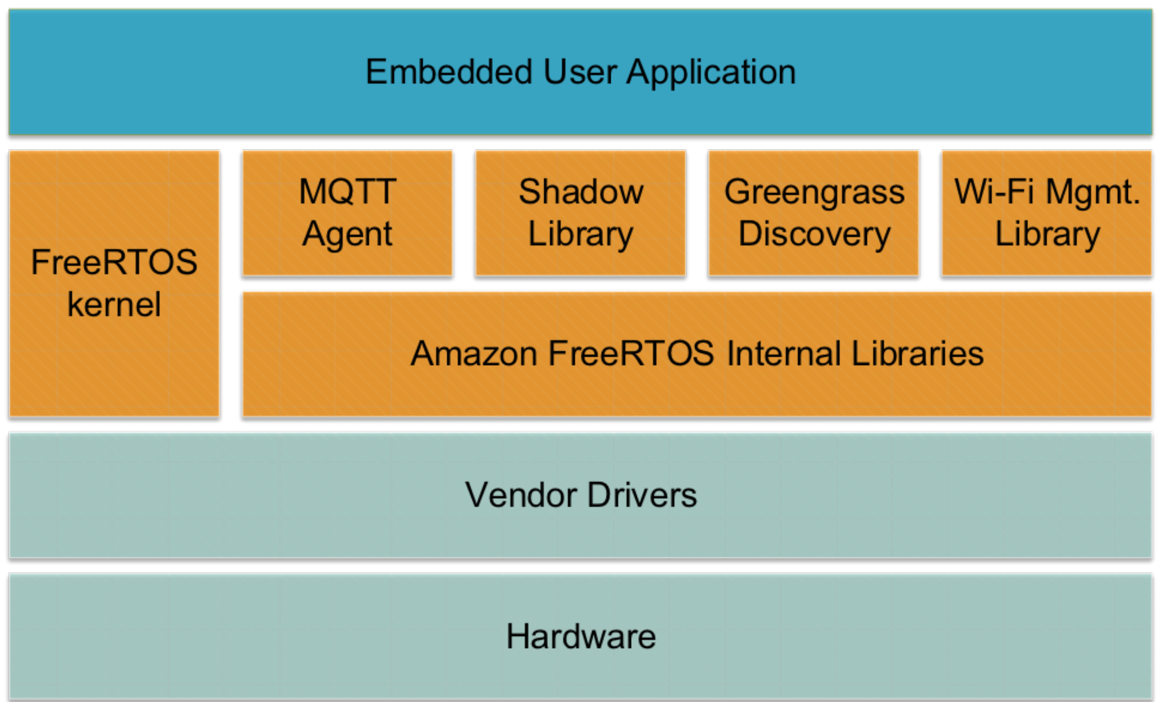
This section contains information required for writing embedded applications with Amazon FreeRTOS.

Topics

- Amazon FreeRTOS Architecture (p. 40)
- FreeRTOS Kernel Fundamentals (p. 41)
- FreeRTOS Libraries (p. 46)
- Amazon FreeRTOS Over the Air Updates (p. 56)
- Amazon FreeRTOS Configuration Wizard User Guide (p. 90)
- Downloading Amazon FreeRTOS from GitHub (p. 91)
- Amazon FreeRTOS Qualification Program (p. 91)
- Supported Platforms (p. 92)

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.



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 (p. 46)
- Greengrass Connectivity (p. 48)
- Amazon FreeRTOS Security (p. 50)
- FreeRTOS Wi-Fi Interface (p. 54)
- OTA agent Library (p. 55)

Cloud Connectivity

Topics

- MQTT (p. 46)
- Device Shadows (p. 47)

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. 46). 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 agent. The caller provides messaging protocol context, an optional callback, and a timeout.

`OTA_AgentShutdown`

Cleans up resources after using the OTA agent.

`OTA_GetAgentState`

Gets the current state of the OTA agent.

`OTA_ActivateNewImage`

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

`OTA_SetImageState`

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

`OTA_GetImageState`

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

`OTA_CheckForUpdate`

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

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

1. Connect to the AWS IoT MQTT broker. For more information, see MQTT (p. 46).
2. Initialize the OTA agent by calling `OTA_AgentInit`. Your application may define a custom OTA callback function or use the default callback by specifying a NULL callback function pointer. You must also supply an initialization timeout.

   The callback implements application-specific logic that executes after completing an OTA update job. The timeout defines how long to wait for the initialization to complete.
3. If `OTA_AgentInit` timed out before the agent was ready, you can call `OTA_GetAgentState` to confirm that the agent is initialized and operating as expected.
4. When the OTA update is complete, Amazon FreeRTOS calls the job completion callback with one of the following events: `accepted`, `rejected`, or `self test`.
5. If the new firmware image has been rejected (for example, due to a validation error), the application can typically ignore the notification and wait for the next update.
6. If the update is valid and has been marked as accepted, call `OTA_ActivateNewImage` to reset the device and boot the new firmware image.

Amazon FreeRTOS Over the Air Updates

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

When you create an OTA update, the OTA Update Manager Service (p. 85) creates an AWS IoT job to notify your devices that an update is available. The OTA demo application runs on your device and creates a Amazon FreeRTOS task that subscribes to notification topics for AWS IoT jobs and listens for update messages. When an update is available, the OTA agent publishes requests to AWS IoT streaming topics and receives file blocks using the MQTT protocol. It reassembles the blocks into files and checks the digital signature of the downloaded files. If the files are valid, it installs the firmware update. If you are not using the Amazon FreeRTOS OTA Update demo application, you must integrate the OTA agent

library into your own application to get the firmware update capability. For more information, see OTA agent Library (p. 55).

Amazon FreeRTOS over the air updates makes it possible for 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 new firmware after it's deployed to devices.
- Monitor the progress of a deployment.
- Debug a failed deployment.

OTA is supported in the following regions:

- `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 selected region is displayed in the upper-right 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.

Over the Air Update Prerequisites

To use over the air updates, you need to:

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

Create an Amazon S3 Bucket to Store Your Update

OTA update files are stored in Amazon S3 buckets. If you are using Code Signing for Amazon FreeRTOS, the command you use to create a code-signing job takes a source bucket (where the unsigned firmware

image is located) and a destination bucket (where the signed firmware image is written). You can specify the same bucket for both the source and destination. The file names are changed to GUIDs so the original files are not overwritten.

To create an Amazon S3 bucket

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**, choose **Save**, and then choose **Next**.
6. Choose **Next** to accept the default permissions.
7. Choose **Create bucket**.

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

Creating an OTA Update Service Role

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

To create an OTA service role

1. Sign in to the https://console.aws.amazon.com/iam/.
2. From the navigation pane, choose **Roles**.
3. Choose **Create role**.
4. Under **Select type of trusted entity**, choose **AWS Service**.
5. Choose **IoT** from the list of AWS services.
6. Under **Select your use case**, choose **IoT Allows IoT to call AWS services on your behalf**.
7. Choose **Next: Permissions**.
8. Choose **Next: Review**.
9. Type a role name and description, and then choose **Create role**.
10. From the list of roles, choose the role you just created.
11. Choose **Add inline policy**.
12. Choose the **JSON** tab,
13. In the following JSON code, replace *<example_bucket>* with your Amazon S3 bucket name. Replace *<your_account_id>* with your AWS account ID. Replace *<role-name>* with your OTA service role name. Copy and paste the JSON code into the policy editor.

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

```
            "Action": [
                "s3:ListBucket",
                "s3:ListAllMyBuckets",
                "s3:GetBucketLocation"
            ],
            "Resource": "*"
    },
    {
         "Effect": "Allow",
         "Action":"iam:PassRole",
         "Resource":"arn:aws:iam::<your_account_id>:role/<role_name>"
    },
    {
            "Effect": "Allow",
            "Action": [
                "signer:StartSigningJob",
                "signer:PutSigningProfile",
                "signer:GetSigningProfile",
                "signer:DescribeSigningJob",
                "iot:CreateStream",
                "iot:DeleteStream",
                "iot:CreateJob",
                "iot:DeleteJob"
            ],
            "Resource": "*"
    }
    ]
}
```

14. Choose **Review policy**.
15. Enter a name and optional description for your policy.
16. Choose **Create policy**.

For more information about IAM roles, see IAM Roles.

To add OTA update permissions to your OTA service role

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

Creating an OTA User Policy

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

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

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

To create an OTA user policy

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 **Create policy**.
7. Choose the **JSON** tab, and 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:*",
   "freertos:ListHardwarePlatforms"
  ],
  "Resource":"*"
 },
 {
 "Effect":"Allow",
 "Action":[
  "s3:GetObject",
  "s3:PutObject"
 ],
 "Resource":"arn:aws:s3:::<example-bucket>/*"
 },
 {
  "Effect":"Allow",
  "Action":"iam:PassRole",
  "Resource":"arn:aws:iam::<your-account-id>:role/<role-name>"
 }
 ]
}
```

   Replace *<example-bucket>* with the name of the Amazon S3 bucket where your OTA update firmware image is stored. Replace *<your-account-id>* with your AWS account ID. You can find your AWS account ID in the upper right of the console. When you enter your account ID, remove any dashes (-). Replace *<role-name>* with the name of the IAM service role you just created.
8. Choose **Review policy**.
9. Enter a name for your new OTA user policy, and then choose **Create policy**.

To attach the OTA user policy to your IAM user

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

Creating a Code-Signing Certificate

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

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

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

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

- Production (certificate-catalog)

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

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

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

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

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

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

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

To create a self-signed code signing certificate

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

```
[ req ]
prompt             = no
distinguished_name = my dn

[ my dn ]
commonName = test_signer@amazon.com
```

```
[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

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

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

3. Convert the Texas Instruments playground root CA private key from DER format to PEM format.

   The TI playground root CA private key is located here:

   `C:\ti\simplelink_cc32xx_sdk_2_10_00_04\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert-key` (Windows)

   `/Applications/Ti/simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert-key` (macOS)

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

4. Convert the Texas Instruments playground root CA certificate from DER format to PEM format.

   The TI playground root certificate is located here:

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

   `/Applications/Ti/simplelink_cc32xx_sdk_2_10_00_04/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert` (macOS)

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

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

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

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

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

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

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

```
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 Code Signing for Amazon FreeRTOS to sign your firmware images. Although the use of Code Signing for Amazon FreeRTOS is recommended, you can sign your firmware images manually.

Creating a Code-Signing Certificate for the Microchip Curiosity PIC32MZEF

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

1. 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
```

2. 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
```

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

```
aws acm import-certificate --certificate file://ecdsasigner.crt --private-key
        file://ecdsasigner.key
```

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

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

Creating a Code-Signing Certificate for the Amazon FreeRTOS Windows Simulator

The Amazon FreeRTOS Windows simulator requires a code-signing certificate with an RSA-2048 bit key and SHA-256 hash to perform OTA updates. If you don't have a code-signing certificate, use these steps to create one:

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

```
[ req ]
prompt             = no
distinguished_name = my_dn

[ my_dn ]
commonName = test_signer@amazon.com

[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

2. Create a code-signing private key and certificate:

```
openssl req -x509 -config cert_config -extensions my_exts -nodes -days 365 -newkey
 rsa:2048 -keyout rsasigner.key -out rsasigner.crt
```

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

```
aws acm import-certificate --certificate file://rsasigner.crt --private-key file://
rsasigner.key
```

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 Code Signing for Amazon FreeRTOS to sign your firmware images. Although the use of Code Signing for Amazon FreeRTOS is recommended, you can sign your firmware images manually.

Creating a Code-Signing Certificate for Custom Hardware

Using an appropriate toolset, create a self-signed certificate and private key for your hardware.

After you create your code-signing certificate, import it into ACM:

```
aws acm import-certificate --certificate file://code-sign.crt --private-key file://code-
sign.key
```

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

ACM requires certificates to use specific algorithms and key sizes. For more information, see Prerequisites for Importing Certificates. For more information about ACM, see Importing Certificates into AWS Certificate Manager.

You must copy, paste, and format the contents of your code-signing certificate and private key into the `aws_ota_codesigner_certificate.h` file that is part of the Amazon FreeRTOS code you download later.

Granting Access to Code Signing for Amazon FreeRTOS

In production environments, you should digitally sign your firmware update to ensure the authenticity and integrity of the update. You can sign your update manually or you can use Code Signing for Amazon FreeRTOS to sign your code. To use Code Signing for Amazon FreeRTOS, you must grant your IAM user account access to Code Signing for Amazon FreeRTOS.

To grant your IAM user account permissions for Code Signing for Amazon FreeRTOS

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. Enter a policy name and description, and then choose **Create policy**.
7. In the navigation pane, choose **Users**.
8. Choose your IAM user account.
9. On the **Permissions** tab, choose **Add permissions**.
10. Choose **Attach existing policies directly**, and then select the check box next to the code-signing policy you just created.
11. Choose **Next: Review**.
12. Choose **Add permissions**.

Download Amazon FreeRTOS with the OTA Library

Download and Build Amazon FreeRTOS for the Texas Instruments CC3200SF-LAUNCHXL

To download Amazon FreeRTOS and the OTA demo code

1. Browse to the AWS IoT console and from the navigation pane, choose **Software**.
2. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
3. From the list of software configurations, choose **Connect to AWS IoT - TI**. Choose the configuration name, not the **Download** link.

   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 Updates**.
6. Under **Name required**, type **`Connect-to-IoT-OTA-TI`**, and then choose **Create and download**.

Save the zip file that contains Amazon FreeRTOS and the OTA demo code to your computer.

To build the demo application

1. Extract the .zip file.
2. Follow the instructions in Getting Started with Amazon FreeRTOS (p. 4), to import the `aws_demos` project into Code Composer Studio, configure your AWS IoT endpoint, your Wi-Fi SSID and password, and a private key and certificate for your board.
3. Open the projet in Code Composer Studio and build it to make sure it compiles without errors.
4. Start a terminal emulator and use the following settings to connect to your board:
   - Baud rate: 115200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
5. Run the project on your board to make sure it can connect to Wi-Fi and the AWS IoT MQTT message broker.

When run, the terminal emulator should display text like the following:

```
0 0 [Tmr Svc] Simple Link task created
Device came up in Station mode
1 374 [Tmr Svc] Starting key provisioning...
```

```
2 374 [Tmr Svc] Write root certificate...
3 474 [Tmr Svc] Write device private key...
4 574 [Tmr Svc] Write device certificate...
SL Disconnect...
5 666 [Tmr Svc] Key provisioning done...
Device came up in Station mode
Device disconnected from the AP on an ERROR..!!
[WLAN EVENT] STA Connected to the AP: Guest , BSSID: 11:22:a1:b2:c3:d4
[NETAPP EVENT] IP acquired by the device

Device has connected to Guest
Device IP Address is 111.222.3.44

6 3718 [OTA] OTA demo version 0.9.0
7 3719 [OTA] Creating MQTT Client...
8 3719 [OTA] Connecting to broker...
9 3719 [OTA] Sending command to MQTT task.
10 3719 [MQTT] Received message 10000 from queue.
11 4798 [MQTT] MQTT Connect was accepted. Connection established.
12 4798 [MQTT] Notifying task.
13 4799 [OTA] Command sent to MQTT task passed.
14 4799 [OTA] Connected to broker.
15 4800 [OTA Task] Sending command to MQTT task.
16 70 27690 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
71 28690 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
72 29690 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
```

Download and Build Amazon FreeRTOS for the Microchip Curiosity PIC32MZEF

To download the Amazon FreeRTOS OTA demo code

1. Browse to the AWS IoT console and from the navigation pane, choose **Software**.
2. Under **Amazon FreeRTOS Device Software**, choose **Configure download**.
3. From the list of software configurations, choose **Connect to AWS IoT - Microchip**. Choose the configuration name, not the **Download** link.

   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. Open `aws_demos/lib/aws/ota/aws_ota_codesigner_certificate.h`.
4. Paste the contents of your code-signing certificate into the `static const char signingcredentialSIGNING_CERTIFICATE_PEM` variable. Following the same format as `aws_clientcredential_keys.h`, each line must end with the new line character ('\n') and be enclosed in quotation marks.

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

```
"-----BEGIN CERTIFICATE-----\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";
```

5. Install Python 3 or higher.
6. Install `pyOpenSSL` by running `pip install pyopenssl`.
7. Copy your code-signing certificate in pem format in the path `\demos\common\ota\bootloader\utility\codesigner_cert_utility\` Rename the certificate file as aws_ota_codesigner_certificate.pem.
8. Open the project in MPLAB X IDE and build it to make sure it compiles without errors.
9. Start a terminal emulator and use the following settings to connect to your board:

   - Baud rate: 115200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
10. Unplug the debugger from your board and run the project on your board to make sure it can connect to Wi-Fi and the AWS IoT MQTT message broker.

When you run the project, the MPLAB X IDE should open an output window. Make sure the ICD4 tab is selected. You should see the following output.

```
Bootloader version 00.09.00
[prvBOOT_Init] Watchdog timer initialized.
[prvBOOT_Init] Crypto initialized.

[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] No application image or magic code present at: 0xbd000000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd000000

[prvValidateImage] Validating image at Bank : 1
[prvValidateImage] No application image or magic code present at: 0xbd100000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd100000

[prvBOOT_ValidateImages] Booting default image.


>0 36246 [IP-task] vDHCPProcess: offer ac140a0eip
                                                 1 36297 [IP-task] vDHCPProcess: offer
 ac140a0eip
                 2 36297 [IP-task]

IP Address: 172.20.10.14
3 36297 [IP-task] Subnet Mask: 255.255.255.240
4 36297 [IP-task] Gateway Address: 172.20.10.1
5 36297 [IP-task] DNS Server Address: 172.20.10.1


6 36299 [OTA] OTA demo version 0.9.2
7 36299 [OTA] Creating MQTT Client...
8 36299 [OTA] Connecting to broker...
9 38673 [OTA] Connected to broker.
10 38793 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/devthingota/jobs/
$next/get/accepted
```

```
11 38863 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/devthingota/jobs/
notify-next
12 38863 [OTA Task] [OTA_CheckForUpdate] Request #0
13 38964 [OTA] [OTA_AgentInit] Ready.
14 38973 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken:
 0:devthingota ]
15 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
16 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
17 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
18 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
19 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: files
20 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
21 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
22 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
23 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
24 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha256-ecdsa
25 38975 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
26 38975 [OTA Task] [prvOTA_Close] Context->0x8003b620
27 38975 [OTA Task] [prvPAL_Abort] Abort - OK
28 39964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
29 40964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
30 41964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
31 42964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
32 43964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
33 44964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
34 45964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
35 46964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
36 47964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
```

The terminal emulator should display text like the following:

```
AWS Validate: no valid signature in descr: 0xbd000000
AWS Validate: no valid signature in descr: 0xbd100000


>AWS Launch:  No Map performed. Running directly from address: 0x9d000020?
AWS Launch:  wait for app at: 0x9d000020
WILC1000: Initializing...
0 0

>[None] Seed for randomizer: 1172751941
1 0 [None] Random numbers: 00004272 00003B34 00000602 00002DE3
Chip ID 1503a0

[spi_cmd_rsp][356][nmi spi]: Failed cmd response read, bus error...

[spi_read_reg][1086][nmi spi]: Failed cmd response, read reg (0000108c)...

[spi_read_reg][1116]Reset and retry 10 108c

Firmware ver. : 4.2.1

Min driver ver : 4.2.1

Curr driver ver: 4.2.1

WILC1000: Initialization successful!

Start Wi-Fi Connection...
Wi-Fi Connected
2 7219 [IP-task] 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.

Download and Build Amazon FreeRTOS for a Custom Hardware Platform

Download the Amazon FreeRTOS source from GitHub. Create a project in your IDE that includes all required sources and libraries.

Build and run the project to make sure it can connect to AWS IoT.

For more information about porting Amazon FreeRTOS to a new platform, see Amazon FreeRTOS Porting Guide (p. 101).

OTA Tutorial

This section contains a tutorial for updating firmware on devices running Amazon FreeRTOS using OTA updates. You can, however, use OTA updates to send files to any devices connected to AWS IoT.

You can use the AWS IoT console or the AWS CLI to create an OTA update. The console is the easiest way to get started with OTA because it does a lot of the work for you. The AWS CLI is useful when you are automating OTA update jobs, working with a large number of devices, or are using devices that have not been qualified for Amazon FreeRTOS. For more information, see Amazon FreeRTOS Qualification Program.

To create a OTA update

1. Deploy an initial version of your firmware to one or more devices.
2. Verify that the firmware is running correctly.
3. When a firmware update is required, make the code changes and build the new image.
4. If you are manually signing your firmware, sign and then upload the signed firmware image to your Amazon S3 bucket.

   If you are using Code Signing for Amazon FreeRTOS, upload your unsigned firmware image to an Amazon S3 bucket.
5. Create an OTA update.

The Amazon FreeRTOS OTA agent on the device receives the updated firmware image and verifies the digital signature, checksum, and version number of the new image. If the firmware update is verified, the device is reset and, based on application-defined logic, commits the update. If your devices are not running Amazon FreeRTOS, you must implement an OTA agent that runs on your devices.

Installing the Initial Firmware

To update firmware, you must install an initial version of the firmware that uses the OTA agent library to listen for OTA update jobs. If you are not running Amazon FreeRTOS, skip this step. You must copy your OTA agent implementation onto your devices instead.

Topics

- Install the Initial Version of Firmware on the Texas Instruments CC3200SF-LAUNCHXL (p. 71)
- Install the Initial Version of Firmware on the Microchip Curiosity PIC32MZEF (p. 73)
- Initial Firmware on the Windows Simulator (p. 75)
- Install the Initial Version of Firmware on a Custom Board (p. 75)

Install the Initial Version of Firmware on the Texas Instruments CC3200SF-LAUNCHXL

These steps assume you have already built the `aws_demos` project, as described in Download and Build Amazon FreeRTOS for the Texas Instruments CC3200SF-LAUNCHXL (p. 65).

1. On your Texas Instruments CC3200SF-LAUNCHXL, place the SOP jumper on the middle set of pins (position = 1) and reset the board.
2. Download and install the TI Uniflash tool.
3. Start Uniflash and from the list of configurations, choose **CC3220SF-LAUNCHXL**, then choose **Start Image Creator**.
4. Choose **New Project**.
5. On the **Start new project** page, enter a name for your project. for **Device Type**, choose **CC3220SF**. For **Device Mode**, choose **Develop**. Then choose **Create Project**.
6. Disconnect your terminal emulator. On the right side of the Uniflash application window, choose the **Connect** button.
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_2_10_00_04/tools/cc32xx_tools/certificate-playground` directory, select `dummy-root-ca-cert`, choose **Open**, and then choose **Write**.
10. In the **File** selector pane, choose the **Add File** icon .
11. Browse to the working directory where you created the code-signing certificate and private key, choose `tisigner.crt.der`, choose **Open**, and then choose **Write**.
12. From the **Action** drop-down list, choose **Select MCU Image**, and then choose **Browse** to choose the firmware image to use write to your device (**aws_demos.bin**). This file is located in the `AmazonFreeRTOS/demos/ti/cc3200_launchpad/ccs/Debug` directory. Choose **Open**.
13. 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 `tisigner.key` 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**.
14. In the left pane, under **Files**, select **Service Pack**.
15. Under **Service Pack File Name** choose **Browse**, browse to `simplelink_cc32x_sdk_2_10_00_04/tools/servicepack-cc3x20`, choose `sp_3.4.0.0_2.0.0.0_2.2.0.5.bin`, and then choose **Open**.
16. In the left pane, under **Files**, select **Trusted Root-Certificate Catalog**.
17. Clear the **Use default Trusted Root-Certificate Catalog** check box.
18. Under **Source File**, choose **Browse**, select **simplelink_cc32xx_sdk_2_10_00_04/tools/certificate-playground\certcatalogPlayGround20160911.lst**, and then choose **Open**.
19. Under **Signature Source File**, choose **Browse**, select **simplelink_cc32xx_sdk_2_10_00_04/tools/certificate-playground\certcatalogPlayGround20160911.lst.signed.bin**, and then choose **Open**.
20. Choose the button to save your project.
21. Choose the button.
22. Choose **Program Image (Create and Program)**.

23. After the programming process is complete, place the SOP jumper onto the first set of pins (position = 0), reset the board, and reconnect your terminal emulator to make sure the output is the same as when you debugged the demo with Code Composer Studio. Make a note of the application version number in the terminal output. You use this version number later to verify that your firmware has been updated by an OTA update.

    The terminal should display output like the following:

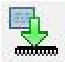
```
0 0 [Tmr Svc] Simple Link task created

Device came up in Station mode

1 369 [Tmr Svc] Starting key provisioning...
2 369 [Tmr Svc] Write root certificate...
3 467 [Tmr Svc] Write device private key...
4 568 [Tmr Svc] Write device certificate...
SL Disconnect...

5 664 [Tmr Svc] Key provisioning done...
Device came up in Station mode

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

[WLAN EVENT] STA Connected to the AP: Guest , BSSID: 11:22:a1:b2:c3:d4

[NETAPP EVENT] IP acquired by the device


Device has connected to Guest

Device IP Address is 111.222.3.44


6 1716 [OTA] OTA demo version 0.9.0
7 1717 [OTA] Creating MQTT Client...
8 1717 [OTA] Connecting to broker...
9 1717 [OTA] Sending command to MQTT task.
10 1717 [MQTT] Received message 10000 from queue.
11 2193 [MQTT] MQTT Connect was accepted. Connection established.
12 2193 [MQTT] Notifying task.
13 2194 [OTA] Command sent to MQTT task passed.
14 2194 [OTA] Connected to broker.
15 2196 [OTA Task] Sending command to MQTT task.
16 2196 [MQTT] Received message 20000 from queue.
17 2697 [MQTT] MQTT Subscribe was accepted. Subscribed.
18 2697 [MQTT] Notifying task.
19 2698 [OTA Task] Command sent to MQTT task passed.
20 2698 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/$next/get/
accepted

21 2699 [OTA Task] Sending command to MQTT task.
22 2699 [MQTT] Received message 30000 from queue.
23 2800 [MQTT] MQTT Subscribe was accepted. Subscribed.
24 2800 [MQTT] Notifying task.
25 2801 [OTA Task] Command sent to MQTT task passed.
26 2801 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/notify-next

27 2814 [OTA Task] [OTA] Check For Update #0
28 2814 [OTA Task] Sending command to MQTT task.
29 2814 [MQTT] Received message 40000 from queue.
30 2916 [MQTT] MQTT Publish was successful.
31 2916 [MQTT] Notifying task.
32 2917 [OTA Task] Command sent to MQTT task passed.
33 2917 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:TI-LaunchPad ]
```

```
34 2917 [OTA Task] [OTA] Missing job parameter: execution
35 2917 [OTA Task] [OTA] Missing job parameter: jobId
36 2918 [OTA Task] [OTA] Missing job parameter: jobDocument
37 2918 [OTA Task] [OTA] Missing job parameter: ts_ota
38 2918 [OTA Task] [OTA] Missing job parameter: files
39 2918 [OTA Task] [OTA] Missing job parameter: streamname
40 2918 [OTA Task] [OTA] Missing job parameter: certfile
41 2918 [OTA Task] [OTA] Missing job parameter: filepath
42 2918 [OTA Task] [OTA] Missing job parameter: filesize
43 2919 [OTA Task] [OTA] Missing job parameter: sig-sha1-rsa
44 2919 [OTA Task] [OTA] Missing job parameter: fileid
45 2919 [OTA Task] [OTA] Missing job parameter: attr
47 3919 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
48 4919 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
49 5919 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
```

Install the Initial Version of Firmware on the Microchip Curiosity PIC32MZEF

These steps assume you have already built the `aws_demos` project, as described in Download and Build Amazon FreeRTOS for the Microchip Curiosity PIC32MZEF (p. 66).

To burn the demo application onto your board

1. Open `aws_demos/lib/aws/ota/aws_ota_codesigner_certificate.h`.
2. 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 enclosed in quotation marks.
3. Rebuild the `aws_demos` project and make sure it compiles without errors.
4. On the tool bar, choose .
5. After the programming process is complete, disconnect the ICD 4 debugger and reset the board. Reconnect your terminal emulator to make sure the output is the same as when you debugged the demo with MPLAB X IDE.

The terminal should display output similar to the following:

```
Bootloader version 00.09.00
[prvBOOT_Init] Watchdog timer initialized.
[prvBOOT_Init] Crypto initialized.

[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] No application image or magic code present at: 0xbd000000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd000000

[prvValidateImage] Validating image at Bank : 1
[prvValidateImage] No application image or magic code present at: 0xbd100000
[prvBOOT_ValidateImages] Validation failed for image at 0xbd100000

[prvBOOT_ValidateImages] Booting default image.


>0 36246 [IP-task] vDHCPProcess: offer ac140a0eip
                                                 1 36297 [IP-task] vDHCPProcess: offer
 ac140a0eip
                2 36297 [IP-task]

IP Address: 172.20.10.14
```

```
3 36297 [IP-task] Subnet Mask: 255.255.255.240
4 36297 [IP-task] Gateway Address: 172.20.10.1
5 36297 [IP-task] DNS Server Address: 172.20.10.1


6 36299 [OTA] OTA demo version 0.9.2
7 36299 [OTA] Creating MQTT Client...
8 36299 [OTA] Connecting to broker...
9 38673 [OTA] Connected to broker.
10 38793 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/devthingota/jobs/
$next/get/accepted
11 38863 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/devthingota/jobs/
notify-next
12 38863 [OTA Task] [OTA_CheckForUpdate] Request #0
13 38964 [OTA] [OTA_AgentInit] Ready.
14 38973 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken:
 0:devthingota ]
15 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
16 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
17 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
18 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
19 38973 [OTA Task] [prvParseJSONbyModel] parameter not present: files
20 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
21 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
22 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
23 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
24 38975 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha256-ecdsa
25 38975 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
26 38975 [OTA Task] [prvOTA_Close] Context->0x8003b620
27 38975 [OTA Task] [prvPAL_Abort] Abort - OK
28 39964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
29 40964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
30 41964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
31 42964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
32 43964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
33 44964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
34 45964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
35 46964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
36 47964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
```

Building and flashing factory image

1. Make sure you have SRecord tools installed from Source Forge and verify that `srec_cat` and `srec_info` are in your system path.
2. Update the OTA sequence number and application version for the factory image.
3. Build the project aws_demos.
4. Run the utility script factory_image_generator.py to generate the factory image.

```
factory_image_generator.py -b mplab.production.bin -p MCHP-Curiosity-PIC32MZEF -k
 private_key.pem  -x aws_bootloader.X.production.hex
```

This command takes the following parameters:

- `mplab.production.bin` - the application binary (this is not OTA binary)
- `MCHP-Curiosity-PIC32MZEF` - the platform name
- `private_key.pem` - the code signing private key
- `aws_bootloader.X.production.hex` - the bootloader hex file

The generated unified hex file be named `mplab.production.unified.factory.hex`.

5. Program the generated hex file using MPlab's IPE tool on the device.

Initial Firmware on the Windows Simulator

When using the Windows simulator there is no need to flash an initial version of the firmware. The Windows simulator is part of the aws_demos application which also includes the "firmware".

Install the Initial Version of Firmware on a Custom Board

Using your IDE, build the `aws_demos` project, making sure to include the OTA library. For more information about the structure of the Amazon FreeRTOS source code, see Navigating the Demo Applications (p. 94).

Make sure to include your code-signing certificate, private key, and certificate trust chain either in the Amazon FreeRTOS project or on your device.

Using the appropriate tool, burn the application onto your board and make sure it is running correctly.

Update the Version of Your Firmware

The OTA agent included with Amazon FreeRTOS checks the version of any update and installs it only if it is more recent than the existing firmware version. The following steps show you how to increment the firmware version of the OTA demo application.

1. Open the `aws_demos` project in your IDE.
2. Open `aws_demos/application_code/common_demos/include/aws_application_version.h` and increment the `APP_VERSION_BUILD` token value to 1.
3. If you are using the Microchip Curiosity PIC32MZEF, increment the OTA sequence number in `\demos\common\ota\bootloader\utility\user-config\ota-descriptor.config`. The OTA sequence number should be incremented for every new OTA image generated.
4. Rebuild the project.

The complete version of the firmware is now be 0.9.1. You must copy your firmware update into the Amazon S3 bucket that you created as described in Create an Amazon S3 Bucket to Store Your Update (p. 57). The name of the file you need to copy to Amazon S3 depends on the hardware platform you are using:

- Texas Instruments CC3200SF-LAUNCHXL : `demos\ti\cc3220_launchpad\ccs\debug\aws_demos.bin`
- Microchip Curiosity PIC32MZEF : `demos\microchip\curiosity_pic32mzef\mplab\dist\pic32mz_ef_curiosity\production\mplab.production.ota.bin`

Creating an OTA Update (AWS IoT Console)

1. In the navigation pane of the AWS IoT console, choose **Manage**, and then choose **Jobs**.
2. Choose **Create**.
3. Under **Create an Amazon FreeRTOS Over-the-Air (OTA) update job**, choose **Create OTA update job**.
4. You can deploy an OTA update to a single device or a group of devices. Under **Select devices to update**, choose **Select**. To update a single device, choose the **Things** tab. To update a group of devices, choose the **Thing Groups** tab.
5. If you are updating a single device, select the check box next to the IoT thing associated with your device. If you are updating a group of devices, select the check box next to the thing group associated with your devices. Choose **Next**.

6. Under **Select and sign your firmware image**, select **Sign a new firmware image for me**.
7. Under **Device hardware platform**, select your hardware platform.

   Note
   Only hardware platforms that have been qualified for Amazon FreeRTOS are displayed in this drop-down list. If you are using a non-qualified platform, you must use the CLI to create the OTA update. For more information, see Creating an OTA Update with the AWS CLI (p. 77).
8. Under **Select your firmware image in S3 or upload it**, choose select. A list of your Amazon S3 buckets is displayed. Choose the bucket that contains your firmware update and then choose your firmware update within the bucket.
9. Under **Code signing certificate**, choose **Select** to select a previously imported certificate or **Import** to import a new certificate.
10. Under **Pathname of code signing certificate on device**, type the fully qualified path name to the code-signing certificate on your device. This will likely vary by platform.

    Note
    When running on the Microchip Curiosity PIC32MZEF, the code-signing certificate will first be searched for by name within the certificate store and will attempt to use a built-in certificate if not found.

    Important
    On the Texas Instruments CC3220SF-LAUNCHXL, you must not include a leading slash character (**/**) in front of the file name if your code signing certificate exists in the root of the file system on this platform. The OTA update will fail during authentication with a `file not found` error.
11. Under **Pathname of firmware image on device**, type the fully qualified path name to the location where the firmware image will be copied onto your device. This location will also vary by platform.

    Important
    On the Texas Instruments CC3220SF-LAUNCHXL, the firmware image path name must be `/sys/mcuflashimg.bin` due to security restrictions.
12. Under **IAM role for OTA update job**, choose a role that allows access to your S3 bucket, and has the following policies:
    - AWSIoTThingsRegistration
    - AWSIoTOTAUpdate
13. Choose **Next.**
14. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.
15. Enter an ID and a description for your OTA update job and choose **Create**.

Using a previously signed firmware image

1. Under **Select and sign your firmware image**, choose **Select a previously signed firmware image**.
2. Under **Pathname of firmware image on device**, type the fully qualified path name to the location where the firmware image will be copied onto your device. This might be different on different platforms.
3. Under **Previous code signing job**, choose **Select**, and then choose the previous code-signing job used to sign the firmware image you are using for the OTA update.

Using a custom signed firmware image

1. Under **Select and sign your firmware image**, choose **Use my custom signed firmware image**.

2. Under **Pathname of code signing certificate on device**, type the fully qualified path name to the code-signing certificate on your device. This might be different for different platforms.
3. Under **Pathname of firmware image on device**, type the fully qualified path name to the location where the firmware image will be copied onto your device. This might be different on different platforms.
4. Under **Signature**, paste your PEM format signature.
5. Under **Original hash algorithm**, select the hash algorithm used when creating your file signature.
6. Under **Original encryption algorithm**, select the algorithm used when creating your file signature.
7. Under **Select your firmware image in Amazon S3**, select the Amazon S3 bucket and the signed firmware image in the Amazon S3 bucket.

After you have specified the code-signing information, specify the OTA update job type, service role, and an ID for your update.

Note
Do not use any personally identifiable information in the job ID for your OTA update. Examples of personally identifiable information include:

- Your name.
- Your IP address.
- Your email address.
- Your location.
- Bank details.
- Medical information.

1. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.
2. Under **IAM role for OTA update job**, choose your OTA service role.
3. Enter an alphanumeric ID for your job and then choose **Create**.

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

Note
The AWS IoT console does not update the state of jobs automatically. Refresh your browser to see updates.

Connect your serial UART terminal to your device. You should see output that indicates the device is downloading the updated firmware.

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

For a complete walkthrough of how to use the console to create an OTA update, see OTA Demo Application (p. 97).

Creating an OTA Update with the AWS CLI

To create an OTA update with the AWS CLI you:

1. Digitally sign your firmware image.
2. Create a stream of your digitally signed firmware image.
3. Start an OTA update job.

Digitally Signing your Firmware Update

When using the CLI to perform OTA updates you can use Code Signing for Amazon FreeRTOS or sign your firmware update yourself.

Signing your Firmware Image with Code Signing for Amazon FreeRTOS

To sign your firmware image using Code Signing for Amazon FreeRTOS, you must install the Code Signing Tools. Download the tools and read the README file for installation instructions. For more information about Code Signing for Amazon FreeRTOS, see Code Signing for Amazon FreeRTOS. After installing and configuring the code signing tools, copy your unsigned firmware image to your Amazon S3 bucket and start a code signing job with the following CLI command:

```
aws signer start-signing-job --source 's3={bucketName=<your-source-bucket-name>,
key=<your-image-file-name>, version=<your-s3-file-version>}' --destination
 's3={bucketName=<your-destination-bucket-name>}' --signing-material
 "certificateArn=arn:aws:acm::<your-region>:<your-aws-account-id>:certificate/<your-
certificate-id>" --signing-parameters certname=<cert.pem> --platform <your-hardware-
platform>
```

> **Note**
> *<your-source-bucket-name>* and *<your-destination-bucket-name>* can be the same Amazon S3 bucket.

The following text describes the parameters for the `start-signing-job` command:

`source`

Specifies the location of the unsigned firmware in an S3 bucket.

- `bucketName` - the name of your S3 bucket.
- `key` - the key (file name) of your firmware in your S3 bucket.
- `version` - the S3 version of your firmware in your S3 bucket. This is different from your firmware version and can be found by browsing to the Amazon S3 console, choosing your bucket, and on the top of the console screen next to **Versions**, choosing **Show**.

`destination`

Specifies the destination for the signed firmware in an S3 bucket. The format of this parameter is the same as the `source` parameter.

`signing-material`

The ARN of your code signing certificate. This ARN is generated when you import your certificate into ACM.

`signing-parameters`

A map of key-value pairs for signing. These can include any information that you want to use during signing.

`platform`

The hardware platform to which you are distributing the OTA update. Valid values are:

- `TexasInstruments`
- `MicrochipTechnologyInc`
- `WindowsSimulator`

The signing job will start and write the signed firmware image into the destination Amazon S3 bucket. The file name for the signed firmware image will be a GUID. You will need this file name when creating a

stream. You can find the generated file name by browsing to the Amazon S3 console and choosing your bucket. If you don't see a file with a GUID file name, refresh your browser.

The command will display a job ARN and a job ID. You will need these values later on. For more information about Code Signing for Amazon FreeRTOS, see Code Signing for Amazon FreeRTOS.

Signing your Firmware Image Manually

Digitally sign your firmware image and upload your signed firmware image into your Amazon S3 bucket.

Creating a Stream of your Firmware Update

The OTA Update service sends updates over MQTT messages. To do this you must create a stream that contains your signed firmware update. To do this create a JSON file (stream.json) that identifies your signed firmware image. The JSON file should contain the following:

```
[
{
    "fileId":<your_file_id>,
 "s3Location":{
     "bucket":"<your_bucket_name>",
     "key":"<your_s3_object_key<"
 }
}
]
```

The following list describes the attributes in the JSON file.

`fileId`

An arbitrary integer between 0 - 255 that identifies your firmware image.

`s3Location`

The bucket and key for the firmware to stream.

`bucket`

The Amazon S3 bucket where your unsigned firmware image is stored.

`key`

The file name of your signed firmware image in the Amazon S3 bucket. You can find this value in the Amazon S3 console by looking at the contents of your bucket. If you are using Code Signing for Amazon FreeRTOS the file name will be a GUID generated by Code Signing for Amazon FreeRTOS.

Use the `create-stream` CLI command to create a stream:

```
aws iot create-stream --stream-id <your_stream_id> --description <your_description> --files
 file://<stream.json> --role-arn <your_role_arn>
```

The following list describes the arguments for the `create-stream` CLI command.

`stream-id`

An arbitrary string to identify the stream.

`description`

An optional description of the stream.

files

One or more references to JSON files that contain data about firmware images to stream. The json file must contain the following attributes:

fileId

An arbitrary file ID.

s3Location

Contains the bucket name where the signed firmware image is stored and the key (file name) of the signed firmware image.

bucket

The Amazon S3 bucket where the signed firmware image is stored.

key

The key (file name) of the signed firmware image. When using Code Signing for Amazon FreeRTOS this will be a GUID.

The following is an example stream.json file:

```
[
{
    "fileId":123,
 "s3Location":{
     "bucket":"codesign-ota-bucket",
     "key":"48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
 }
}
]
```

role-arn

An IAM role that grants access to the Amazon S3 bucket

To find the Amazon S3 object key of your signed firmware image, use the aws signer describe-signing-job --job-id *<my-job-id>* command where my-job-id is the job ID displayed by the create-signing-job CLI command. The output of the describe-signing-job command will contain the key of the signed firmware image. For example:

```
... text deleted for brevity ...
 "signedObject": {
        "s3": {
            "bucketName": "ota-bucket",
            "key": "7309da2c-9111-48ac-8ee4-5a4262af4429"
        }
    }
... text deleted for brevity ...
```

Creating an OTA Update

Use the create-ota-update CLI command to create an OTA update job:

```
aws iot create-ota-update --ota-update-id "<my_ota_update>" --target-selection SNAPSHOT
 --description "<a cli ota update>" --files file://<ota.json> --targets arn:aws:iot:<your-
aws-region>:<your-aws-account>:thing/<your-thing-name> --role-arn arn:aws:iam::<your-aws-
account>:role/<your-ota-service-role
```

Note
Do not use any personally identifiable information (PII) in your OTA update job ID. Examples of personally identifiable information include:

- Your name
- Your IP address
- Your email address
- Your location
- Bank details
- Medical information

`ota-update-id`

An arbitrary OTA update ID.

`target-selection`

Valid values are:

- `SNAPSHOT`: The job will terminate after deploying the update to the selected IoT thing or groups.
- `CONTINUOUS`: The job will continue to deploy updates to devices added to the selected groups.

`description`

A text description of the OTA update.

`files`

One or more references to JSON files that contain data about the OTA update. The JSON file can contain the following attributes:

- `fileName`: The fully qualified firmware image file name. For Texas Instruments CC3200SF-LAUNCHXL this must be `"/sys/mcuflashimg.bin"`. For Microchip this must be `"mplab.production.bin"`
- `fileSource`: Contains information about the firmware update stream.
  - `streamId`: The stream ID specified in the `create-stream` CLI command.
  - `fileId`: The file ID specified in the JSON file passed to `create-stream`.
- `codeSigning`: Contains information about the code signing job.
  - `awsSignerJobId`: The signer job ID generated by the `start-siging-job` command.
  - `customCodeSigning`: Contains information about a custom signature.
    - `signature`: Contains a custom signature.
      - `inlineDocument`: The custom signature.
    - `certificateChain`: Contains a certificate chain for a custom signature.
      - `inlineDocument`: The certificate chain.
    - `hashAlgorithm`: The hash algorithm used to create the signature.
    - `signatureAlgorithm`: The signature algorithm used for code signing.
  - `attributes`: Arbitrary key/value pairs.

`targets`

One or more IoT thing ARNs that specify which devices will be updated by the OTA update.

`role-arn`

Your service role's ARN.

The following is an example of a JSON file passed into the `create-ota-update` command that uses Code Signing for Amazon FreeRTOS :

```
[
{
    "fileName": "firmware.bin",
 "fileSource": {
     "streamId": "004",
     "fileId":123
 },
 "codeSigning": {
  "awsSignerJobId": "48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
 }
}
]
```

The following is an example of a JSON file passed into the `create-ota-update` CLI command that uses an inline file to provide custom code signing material:

```
[
{
    "fileName": "firmware.bin",
 "fileSource": {
     "streamId": "004",
     "fileId": 123
 },
 "codeSigning": {
     "customCodeSigning":{
            "signature":{
             "inlineDocument":"<your_signature>"
         },
         "certificateChain": {
             "inlineDocument":"<your_certificate_chain>"
         },
         "hashAlgorithm":"<your_hash_algorithm>",
         "signatureAlgorithm":"<your_sig_algorithm>"
     }
 }
}
]
```

You can use the `get-ota-update` CLI command to get the status of an OTA update:

```
aws iot get-ota-update --ota-update-id <your-ota-update-id>
```

This will return one of the following values:

`CREATE_PENDING`

`CREATE_IN_PROGRESS`

`CREATE_COMPLETE`

`CREATE_FAILED`

Listing OTA Updates

You can get a list of all OTA updates by using the `list-ota-updates` CLI command. For example:

```
aws iot list-ota-updates
```

The output from the `list-ota-updates` command will look like this:

```
{
    "otaUpdates": [

        {
            "otaUpdateId": "my_ota_update2",
            "otaUpdateArn": "arn:aws:iot:us-west-2:123456789012:otaupdate/my_ota_update2",
            "creationDate": 1522778769.042
        },
        {
            "otaUpdateId": "my_ota_update1",
            "otaUpdateArn": "arn:aws:iot:us-west-2:123456789012:otaupdate/my_ota_update1",
            "creationDate": 1522775938.956
        },
        {
            "otaUpdateId": "my_ota_update",
            "otaUpdateArn": "arn:aws:iot:us-west-2:123456789012:otaupdate/my_ota_update",
            "creationDate": 1522775151.031
        }
    ]
}
```

Getting Information About a Specific OTA Update

You can get information about a specific OTA update by using the `get-ota-update` CLI command. For example:

```
aws iot get-ota-update --ota-update-id <my-ota-update-id>
```

The output from the `get-ota-update` command will look like this:

```
{
    "otaUpdateInfo": {
        "otaUpdateId": "myotaupdate1",
        "otaUpdateArn":
        "arn:aws:iot:us-west-2:123456789012:otaupdate/my_ota_update",
        "creationDate": 1522444438.424,
        "lastModifiedDate": 1522444440.681,
        "description": "a test OTA update",
        "targets": [
            "arn:aws:iot:us-west-2:123456789012:thing/myDevice"
        ],
        "targetSelection": "SNAPSHOT",
        "otaUpdateFiles": [
            {
                "fileName": "app.bin",
                "fileSource": {
                    "streamId": "003",
                    "fileId": 123
                },
                "codeSigning": {
                    "awsSignerJobId": "592932bb-24a1-4f91-8ddd-66145352ad19",
                    "customCodeSigning": {}
                }
            }
        ],
        "otaUpdateStatus": "CREATE_COMPLETE",
        "awsIotJobId": "f76da3c0_10eb_41df_9029_ba7abc20f609",
        "awsIotJobArn": "arn:aws:iot:us-
west-2:123456789012:job/f76da3c0_10eb_41df_9029_ba7abc20f609"
    }
}
```

Deleting OTA-Related Data

Currently the AWS IoT console does not allow you to delete streams or OTA updates. You can use the AWS CLI to delete streams, OTA updates, and the IoT jobs created during an OTA update.

Delete an OTA Stream

When you create an OTA update either you or the AWS IoT console creates a stream to break the firmware up into chunks so it can be sent over MQTT. You can delete this stream with the `delete-stream` CLI command. For example:

```
aws iot delete-stream --stream-id <your_stream_id>
```

Deleting an OTA Update

When you create an OTA update several things are created:

- An entry in the OTA update job database.
- An AWS IoT job to perform the update.
- An AWS IoT job execution for each device being updated.

The `delete-ota-update` command deletes the entry in the OTA update job database only. You will still need to delete the AWS IoT job with the `delete-job` command explained below.

To delete an OTA update use the `delete-ota-update` command:

```
aws iot delete-ota-update --ota-update-id <your_ota_update_id>
```

Deleting an IoT Job Created for an OTA Update

The Amazon FreeRTOS service creates an AWS IoT job when you create an OTA update. A job execution is also created for each device that processes the job. You can delete a job and it's associated job executions using the `delete-job` CLI command. For example:

```
aws iot delete-job --job-id <your-job-id --no-force
```

The `no-force` parameter specifies that only jobs that are in a terminal state ("COMPLETED" or "CANCELLED") may be deleted. You can delete a job that is in a non-terminal state by passing the `force` parameter. For more information on the `DeleteJob` API, see DeleteJob API.

> **Note**
> Deleting a job which is "IN_PROGRESS" will interrupt any job executions which are "IN_PROGRESS" on your devices, and may result in a device being left in a non-deterministic state. Use caution and ensure that each device executing a job which is deleted is able to recover to a known state.

Deleting a job may take some time, depending on the number of job executions created for the job and various other factors. While the job is being deleted, the status of the job will be shown as "DELETION_IN_PROGRESS". Attempting to delete or cancel a job whose status is already "DELETION_IN_PROGRESS" will result in an error.

You can delete an individual job execution using the `delete-job-execution`. You may want to delete a job execution when a small number of devices are unable to process a job. This will delete the job execution for a single device. For example:

```
aws iot delete-job-execution --job-id <your-job-id --thing-name
```

```
    <your-thing-name> --execution-number
<your-job-execution-number> --no-force
```

As with the `delete-job` CLI command, you can pass the `--force` parameter to the `delete-job-execution` to force the deletion of an execution job execution. For more information on the `DeleteJobExecution` API, see DeleteJobExecution API.

> **Note**
> Deleting a job execution which is "IN_PROGRESS" will interrupt any job executions which are "IN_PROGRESS" on your devices, and may result in a device being left in a non-deterministic state. Use caution and ensure that each device executing a job which is deleted is able to recover to a known state.

For more information about using the OTA update demo application, see OTA Demo Application (p. 97).

OTA Update Manager Service

The OTA update manager service enables you to create and manage OTA updates. It provides a way to create an OTA update, get information about an existing OTA update, list all OTA updates associated with your AWS account, and delete an OTA update.

An OTA update is a data structure maintained by the OTA update manager service. It contains:

- An OTA update ID.
- An optional OTA update description.
- A list of devices to update (targets).
- The type of OTA update: CONTINUOUS or SNAPSHOT.
- A list of files to send to the target devices.
- An IAM role that allows access to the AWS IoT Jobs service.
- An optional list of user-defined name/value pairs.

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

Once your files have been digitally signed, you create a stream using the Amazon Streaming service which breaks up your files into blocks that can be sent over MQTT to your devices.

When you create an OTA update, the OTA Manager service creates an AWS IoT Job to notify your devices an update is available. The Amazon FreeRTOS OTA Agent runs on your devices and listens for update messages. When an update is available, it streams the update over MQTT and stores the files locally. It checks the digital signature of the downloaded files and if valid, installs the firmware update. If you are not using Amazon FreeRTOS, you will have to implement your own OTA agent to listen for, download updates, and perform any installation operations.

OTA Security

The following are three aspects of OTA security:

Connection security

The OTA Update Manager relies on existing security mechanisms like TLS mutual authentication, used by AWS IoT. OTA update traffic passes through the AWS IoT device gateway and uses AWS

IoT security mechanisms. Each incoming and outgoing MQTT message through the device gateway undergoes strict authentication and authorization.

Authenticity and integrity of OTA updates

Firmware can be digitally signed before an OTA update to ensure that it is from a reliable source and has not been tampered with. The Amazon FreeRTOS OTA Update Manager uses the Code Signing for Amazon FreeRTOS to automatically sign your firmware. For more information, see Code Signing for Amazon FreeRTOS. The OTA agent, which runs on your devices, performs integrity checks on the firmware when it arrives on the device.

Operator security

Every API call made through the control plane API undergoes standard IAM Signature Version 4 authentication and authorization. To create a deployment, you must have permissions to invoke the `CreateDeployment`, `CreateJob`, and `CreateStream` APIs. In addition, in your Amazon S3 bucket policy or ACL, you must give read permissions to the AWS IoT service principal so that the firmware update stored in Amazon S3 can be accessed during streaming.

Code Signing for Amazon FreeRTOS

The AWS IoT console uses Code Signing for Amazon FreeRTOS to automatically sign your firmware image for any device supported by AWS IoT.

Code Signing for Amazon FreeRTOS uses a certificate and private key that you import into ACM. You can use a self–signed certificate for testing, but we recommend that you obtain a certificate from a well–known commercial certificate authority (CA).

Code–signing certificates use the X.509 version 3 **Key Usage** and **Extended Key Usage** extensions. The **Key Usage** extension is set to `Digital Signature` and the **Extended Key Usage** extension is set to `Code Signing`. For more information about signing your code image, see the Code Signing for Amazon FreeRTOS Developer Guide and the Code Signing for Amazon FreeRTOS API Reference.

Note
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.

Setting Up Cloudwatch Logs for OTA Updates

The OTA Update service supports logging with Amazon CloudWatch. You can use the AWS IoT console to enable and configure Amazon CloudWatch logging for OTA updates. For more information about CloudWatch Logs, see Cloudwatch Logs.

To enable logging, you must create an IAM role and configure OTA update logging.

Note
Before you enable OTA update logging, make sure you understand the CloudWatch Logs access permissions. Users with access to CloudWatch Logs can see your debugging information. For information, see Authentication and Access Control for Amazon CloudWatch Logs.

Create a Logging Role and Enable Logging

Use the AWS IoT console to create a logging role and enable logging.

1. From the navigation pane, choose **Settings**.
2. Under **Logs**, choose **Edit**.
3. Under **Level of verbosity**, choose **Debug**.
4. Under **Set role**, choose **Create new** to create an IAM role for logging.

5. Under **Name**, enter a unique name for your role. Your role will be created with all required permissions.
6. Choose **Update**.

OTA Update Logs

The OTA Update service publishes logs to your account when one of the following occurs:

- An OTA update is created.
- An OTA update is completed.
- A code-signing job is created.
- A code-signing job is completed.
- An AWS IoT job is created.
- An AWS IoT job is completed.
- A stream is created.

You can view your logs in the CloudWatch console.

Viewing OTA Update CloudWatch Logs

1. From the navigation pane, choose **Logs**.
2. In **Log Groups**, choose **AWSIoTLogsV2**.

OTA update logs can contain the following properties:

accountId

The AWS account ID in which the log was generated.

actionType

The action that generated the log. This can be set to one of the following values:

- CreateOTAUpdate: An OTA update was created.
- DeleteOTAUpdate: An OTA update was deleted.
- StartCodeSigning: A code-signing job was started.
- CreateAWSJob: An AWS IoT job was created.
- CreateStream: A stream was created.
- GetStream: A request for a stream was sent to the AWS IoT Streaming service.
- DescribeStream: A request for information about a stream was sent to the AWS IoT Streaming service.

awsJobId

The AWS IoT job ID that generated the log.

clientId

The MQTT client ID that made the request that generated the log.

clientToken

The client token associated with the request that generated the log.

details

Additional information about the operation that generated the log.

logLevel

The logging level of the log. This is always set to DEBUG in OTA Update logs.

otaUpdateId

The ID of the OTA update that generated the log.

protocol

The protocol used to make the request that generated the log.

status

The status of the operation that generated the log. Valid values are:

- Success
- Failure

streamId

The AWS IoT stream ID that generated the log.

timestamp

The time when the log was generated.

topicName

An MQTT topic used to make the request that generated the log.

Example Logs

The following is an example log generated when a code-signing job is started:


```
{
    "timestamp": "2018-07-23 22:59:44.955",
    "logLevel": "DEBUG",
    "accountId": "875157236366",
    "status": "Success",
    "actionType": "StartCodeSigning",
    "otaUpdateId": "08957b03-eea3-448a-87fe-743e6891ca3a",
    "details": "Start code signing job. The request status is SUCCESS."
}
```


The following is an example log generated when an AWS IoT job is created:


```
{
    "timestamp": "2018-07-23 22:59:45.363",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "actionType": "CreateAWSJob",
    "otaUpdateId": "08957b03-eea3-448a-87fe-743e6891ca3a",
    "awsJobId": "08957b03-eea3-448a-87fe-743e6891ca3a",
    "details": "Create AWS Job The request status is SUCCESS."
}
```


The following is an example log generated when an OTA update is created:


```
{
```

```
    "timestamp": "2018-07-23 22:59:45.413",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "actionType": "CreateOTAUpdate",
    "otaUpdateId": "08957b03-eea3-448a-87fe-743e6891ca3a",
    "details": "OTAUpdate creation complete. The request status is SUCCESS."
}
```

The following is an example log generated when a stream is created:

```
{
 "timestamp": "2018-07-23 23:00:26.391",
 "logLevel": "DEBUG",
 "accountId": "123456789012",
 "status": "Success",
 "actionType": "CreateStream",
 "otaUpdateId": "3d3dc5f7-3d6d-47ac-9252-45821ac7cfb0",
 "streamId": "6be2303d-3637-48f0-ace9-0b87b1b9a824",
 "details": "Create stream. The request status is SUCCESS."
}
```

The following is an example log generated when an OTA update is deleted:

```
{
 "timestamp": "2018-07-23 23:03:09.505",
 "logLevel": "DEBUG",
 "accountId": "123456789012",
 "status": "Success",
 "actionType": "DeleteOTAUpdate",
 "otaUpdateId": "9bdd78fb-f113-4001-9675-1b595982292f",
 "details": "Delete OTA Update. The request status is SUCCESS."
}
```

The following is an example log generated when a device requests a stream from the streaming service:

```
{
 "timestamp": "2018-07-25 22:09:02.678",
 "logLevel": "DEBUG",
 "accountId": "123456789012",
 "status": "Success",
 "actionType": "GetStream",
 "protocol": "MQTT",
 "clientId": "b9d2e49c-94fe-4ed1-9b07-286afed7e4c8",
 "topicName": "$aws/things/b9d2e49c-94fe-4ed1-9b07-286afed7e4c8/streams/1e51e9a8-9a4c-4c50-b005-d38452a956af/get/json",
 "streamId": "1e51e9a8-9a4c-4c50-b005-d38452a956af",
 "details": "The request status is SUCCESS."
}
```

The following is an example log generated when a device calls the `DescribeStream` API:

```
{
 "timestamp": "2018-07-25 22:10:12.690",
 "logLevel": "DEBUG",
 "accountId": "123456789012",
```

```
 "status": "Success",
 "actionType": "DescribeStream",
 "protocol": "MQTT",
 "clientId": "581075e0-4639-48ee-8b94-2cf304168e43",
 "topicName": "$aws/things/581075e0-4639-48ee-8b94-2cf304168e43/streams/71c101a8-
bcc5-4929-9fe2-af563af0c139/describe/json",
 "streamId": "71c101a8-bcc5-4929-9fe2-af563af0c139",
 "clientToken": "clientToken",
 "details": "The request status is SUCCESS."
}
```

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 pre qualified 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 pre qualified platforms:

- TI CC3220SF-LAUNCHXL
- STM32 IoT Discovery Kit
- NXP LPC54018 IoT Module
- Microchip Curiosity PIC32MZEF
- Espressif ESP32-DevKitC
- Espressif ESP3-WROVER-KIT
- FreeRTOS Windows Simulator

The predefined configurations make it possible for you to get started quickly with the supported use cases without thinking about which libraries are required. To use a predefined configuration, browse to the Amazon FreeRTOS console, find the configuration you want to use, and then choose **Download**.

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 pre qualified 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 contact 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 qualification 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.

Espressif ESP32-DevKitC

The Espressif ESP32-DevKitC is a small and convenient development board with ESP32-WROOM-32 module installed, break out pin headers and minimum additional components. Includes USB to serial programming interface, that also provides power supply for the board. Has pushbuttons to reset the board and put it in upload mode. For more information, see ESP32-DevKitC

Espressif ESP32-WROVER-KIT

The ESP-WROVER-KIT V3 development board has dual port USB to serial converter for programming and JTAG interface for debugging. Power supply is provided by USB interface or from standard 5 mm power supply jack. Power supply selection is done with a jumper and may be put on/off with a separate switch. This board has MicroSD card slot, 3.2" SPI LCD screen and dedicated header to connect a camera. It provides RGB diode for diagnostics. Includes 32.768 kHz XTAL for internal RTC to operate it in low power modes.. For more information, see ESP-WROVER-KIT

Amazon FreeRTOS Demo Projects

This section contains resources that are useful after you have a basic understanding of Amazon FreeRTOS. If you haven't already, we recommend that you first read the Getting Started with Amazon FreeRTOS (p. 4).

Topics

- Navigating the Demo Applications (p. 94)
- Device Shadow Demo Application (p. 95)
- Greengrass Discovery Demo Application (p. 97)
- OTA Demo Application (p. 97)

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. 13)
- Getting Started with the NXP LPC54018 IoT Module (p. 16)
- Getting Started with the Microchip Curiosity PIC32MZEF (p. 20)
- Getting Started with the FreeRTOS Windows Simulator (p. 36)

At this point, you have downloaded the Amazon FreeRTOS software, imported it into your IDE, and built the project without errors. The project is already configured to run the Greengrass Connectivity demo. In the AWS IoT console, choose **Test**, and then add a subscription to `freertos/demos/ggd`. The demo publishes a series of messages to the Greengrass core. The messages are also published to AWS IoT, where they are received by the AWS IoT MQTT client.

In the MQTT client, you should see the following strings:

```
Message from Thing to Greengrass Core: Hello world msg #1!
Message from Thing to Greengrass Core: Hello world msg #0!
Message from Thing to Greengrass Core: Address of Greengrass Core
 found! <123456789012>.<us-west-2>.compute.amazonaws.com
```

OTA Demo Application

Amazon FreeRTOS includes a demo application that demonstrates the use of the OTA library. The OTA demo application is located in the demos\common\ota subdirectory.

Before you create an OTA update, read Amazon FreeRTOS Over the Air Updates (p. 56) and complete all prerequisites listed there.

The OTA demo application:

1. Initializes the FreeRTOS network stack and MQTT buffer pool. (See main.c.)
2. Creates a task to exercise the OTA library. (See `vOTAUpdateDemoTask` in aws_ota_update_demo.c.)
3. Creates an MQTT client using `MQTT_AGENT_Create`.
4. Connects to the AWS IoT MQTT broker using `MQTT_AGENT_Connect`.
5. Calls `OTA_AgentInit` to create the OTA task and registers a callback to be used when the OTA task is complete.

After you have created an OTA update job using either the AWS IoT console or the AWS CLI, connect a terminal emulator to see the progress of the OTA update and note any errors generated during the process.

A successful OTA update job will display output like the following. Some lines in the example below have been removed from the listing for brevity.

```
313 267848 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
314 268733 [OTA Task] [OTA] Set job doc parameter [ jobId:
 fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
315 268734 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
316 268734 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
317 268734 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
318 268735 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
319 268735 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
320 268735 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
321 268737 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa:
 Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
322 268737 [OTA Task] [OTA] Job was accepted. Attempting to start transfer.
323 268737 [OTA Task] Sending command to MQTT task.
324 268737 [MQTT] Received message 50000 from queue.
325 268848 [OTA] [OTA] Queued: 2   Processed: 1   Dropped: 0
326 269039 [MQTT] MQTT Subscribe was accepted. Subscribed.
327 269039 [MQTT] Notifying task.
328 269040 [OTA Task] Command sent to MQTT task passed.
329 269041 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/streams/327

330 269848 [OTA] [OTA] Queued: 2   Processed: 1   Dropped: 0
... // Output removed for brevity
346 284909 [OTA Task] [OTA] file token: 74594452
.. // Output removed for brevity
363 301327 [OTA Task] [OTA] file ready for access.
364 301327 [OTA Task] [OTA] Returned buffer to MQTT Client.
365 301328 [OTA Task] Sending command to MQTT task.
366 301328 [MQTT] Received message 60000 from queue.
367 301328 [MQTT] Notifying task.
368 301329 [OTA Task] Command sent to MQTT task passed.
369 301329 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/
streams/327/get
370 301632 [OTA Task] [OTA] Received file block 0, size 1024
371 301647 [OTA Task] [OTA] Remaining: 127
... // Output removed for brevity
508 304622 [OTA Task] Sending command to MQTT task.
509 304622 [MQTT] Received message 70000 from queue.
510 304622 [MQTT] Notifying task.
511 304623 [OTA Task] Command sent to MQTT task passed.
512 304623 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/
streams/327/get
513 304860 [OTA] [OTA] Queued: 47   Processed: 47   Dropped: 83
514 304926 [OTA Task] [OTA] Received file block 4, size 1024
515 304941 [OTA Task] [OTA] Remaining: 82
... // Output removed for brevity
797 315047 [MQTT] MQTT Publish was successful.
798 315048 [MQTT] Notifying task.
799 315048 [OTA Task] Command sent to MQTT task passed.
800 315049 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/
jobs/fe18c7ec_8c31_4438_b0b9_ad55acd9561801 315049 [OTA Task] Sending command to MQTT task.
802 315049 [MQTT] Received message d0000 from queue.
803 315150 [MQTT] MQTT Unsubscribe was successful.
804 315150 [MQTT] Notifying task.
805 315151 [OTA Task] Command sent to MQTT task passed.
806 315152 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

807 315172 [OTA Task] Sending command to MQTT task.
808 315172 [MQTT] Received message e0000 from queue.
```

```
809 315273 [MQTT] MQTT Unsubscribe was successful.
810 315273 [MQTT] Notifying task.
811 315274 [OTA Task] Command sent to MQTT task passed.
812 315274 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

813 315275 [OTA Task] [OTA] Resetting MCU to activate new image.
0 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created

Device came up in Station mode

2 137 [Tmr Svc] Wi-Fi module initialized.
3 137 [Tmr Svc] Starting key provisioning...
4 137 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 339 [Tmr Svc] Write device certificate...
7 436 [Tmr Svc] Key provisioning done...
Device disconnected from the AP on an ERROR..!!

[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 192.168.3.72


8 1443 [Tmr Svc] Wi-Fi connected to AP Guest.
9 1444 [Tmr Svc] IP Address acquired 192.168.3.72
10 1444 [OTA] OTA demo version 0.9.1
11 1445 [OTA] Creating MQTT Client...
12 1445 [OTA] Connecting to broker...
13 1445 [OTA] Sending command to MQTT task.
14 1445 [MQTT] Received message 10000 from queue.
15 2910 [MQTT] MQTT Connect was accepted. Connection established.
16 2910 [MQTT] Notifying task.
17 2911 [OTA] Command sent to MQTT task passed.
18 2912 [OTA] Connected to broker.
19 2913 [OTA Task] Sending command to MQTT task.
20 2913 [MQTT] Received message 20000 from queue.
21 3014 [MQTT] MQTT Subscribe was accepted. Subscribed.
22 3014 [MQTT] Notifying task.
23 3015 [OTA Task] Command sent to MQTT task passed.
24 3015 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/$next/get/
accepted

25 3028 [OTA Task] Sending command to MQTT task.
26 3028 [MQTT] Received message 30000 from queue.
27 3129 [MQTT] MQTT Subscribe was accepted. Subscribed.
28 3129 [MQTT] Notifying task.
29 3130 [OTA Task] Command sent to MQTT task passed.
30 3138 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/notify-next

31 3138 [OTA Task] [OTA] Check For Update #0
32 3138 [OTA Task] Sending command to MQTT task.
33 3138 [MQTT] Received message 40000 from queue.
34 3241 [MQTT] MQTT Publish was successful.
35 3241 [MQTT] Notifying task.
36 3243 [OTA Task] Command sent to MQTT task passed.
37 3245 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:TI-LaunchPad ]
38 3245 [OTA Task] [OTA] Set job doc parameter [ jobId:
 fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
39 3245 [OTA Task] [OTA] Identified job doc parameter [ self_test ]
40 3246 [OTA Task] [OTA] Set job doc parameter [ updatedBy: 589827 ]
```

```
41 3246 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
42 3246 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
43 3247 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
44 3247 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
45 3247 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
46 3247 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
47 3249 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa:
 Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
48 3249 [OTA Task] [OTA] Job is ready for self test.
49 3250 [OTA Task] Sending command to MQTT task.
51 3351 [MQTT] MQTT Publish was successful.
52 3352 [MQTT] Notifying task.
53 3352 [OTA Task] Command sent to MQTT task passed.
54 3353 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/jobs/
fe18c7ec_8c31_4438_b0b9_ad55acd95610/u55 3353 [OTA Task] Sending command to MQTT task.
56 3353 [MQTT] Received message 60000 from queue.
57 3455 [MQTT] MQTT Unsubscribe was successful.
58 3455 [MQTT] Notifying task.
59 3456 [OTA Task] Command sent to MQTT task passed.
60 3456 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

61 3456 [OTA Task] [OTA] Accepted final image. Commit.
62 3578 [OTA Task] Sending command to MQTT task.
63 3578 [MQTT] Received message 70000 from queue.
64 3779 [MQTT] MQTT Publish was successful.
65 3780 [MQTT] Notifying task.
66 3780 [OTA Task] Command sent to MQTT task passed.
67 3781 [OTA Task] [OTA] Published 'SUCCEEDED' status to $aws/things/TI-LaunchPad/jobs/
fe18c7ec_8c31_4438_b0b9_ad55acd95610/upd68 3781 [OTA Task] [OTA] Returned buffer to MQTT
 Client.
69 4251 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
70 4381 [OTA Task] [OTA] Missing job parameter: execution
71 4382 [OTA Task] [OTA] Missing job parameter: jobId
72 4382 [OTA Task] [OTA] Missing job parameter: jobDocument
73 4382 [OTA Task] [OTA] Missing job parameter: ts_ota
74 4382 [OTA Task] [OTA] Missing job parameter: files
75 4382 [OTA Task] [OTA] Missing job parameter: streamname
76 4382 [OTA Task] [OTA] Missing job parameter: certfile
77 4382 [OTA Task] [OTA] Missing job parameter: filepath
78 4383 [OTA Task] [OTA] Missing job parameter: filesize
79 4383 [OTA Task] [OTA] Missing job parameter: sig-sha1-rsa
80 4383 [OTA Task] [OTA] Missing job parameter: fileid
81 4383 [OTA Task] [OTA] Missing job parameter: attr
82 4383 [OTA Task] [OTA] Returned buffer to MQTT Client.
83 5251 [OTA] [OTA] Queued: 2   Processed: 2   Dropped: 0
```

Amazon FreeRTOS Porting Guide

This porting guide walks you through modifying the Amazon FreeRTOS software package to work on boards that are not Amazon FreeRTOS qualified. Amazon FreeRTOS is designed to let you choose only those libraries required by your board or application. The MQTT, Shadow, and Greengrass libraries are designed to be compatible with most devices as-is, so there is no porting guide for these libraries.

For information about porting FreeRTOS kernel, see FreeRTOS Kernel Porting Guide.

Topics

- Bootloader (p. 101)
- Logging (p. 101)
- Connectivity (p. 102)
- Security (p. 103)
- Using Custom Libraries with Amazon FreeRTOS (p. 105)
- OTA Portable Abstraction Layer (p. 105)

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. 50). For information about porting TLS and cryptographic operations, see the TLS (p. 51) and Public Key Cryptography Standard #11 (p. 52) 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.