Table of Contents

What Is Amazon FreeRTOS? ................................................................................................................................. 1
The FreeRTOS Kernel ........................................................................................................................................... 1
Over-the-Air Updates (Beta) ............................................................................................................................. 1
Amazon FreeRTOS Libraries ............................................................................................................................. 1
Amazon FreeRTOS Console .............................................................................................................................. 2
Amazon FreeRTOS Qualification Program ......................................................................................................... 2
Development Workflow .................................................................................................................................. 2
Getting Started with Amazon FreeRTOS .......................................................................................................... 4
Granting FreeRTOS Permissions ....................................................................................................................... 4
Amazon FreeRTOS Prerequisites ....................................................................................................................... 4
Create Your AWS IoT Credentials .................................................................................................................. 4
Getting Started with the Texas Instruments (TI) CC3220SF-LAUNCHXL ....................................................... 6
Setting Up Your Environment .......................................................................................................................... 6
Download and Build Amazon FreeRTOS ......................................................................................................... 7
Run the FreeRTOS Samples ........................................................................................................................... 9
Troubleshooting ........................................................................................................................................... 9
Getting Started with the STMicroelectronics (ST) STM32L4 Discovery Kit IoT node ................................... 10
Setting Up Your Environment .......................................................................................................................... 10
Download and Build Amazon FreeRTOS ......................................................................................................... 10
Run the FreeRTOS Samples ........................................................................................................................... 13
Getting Started with the NXP LPC54018 IoT Module .................................................................................... 13
Setting Up Your Environment .......................................................................................................................... 13
Connecting a JTAG Debugger ........................................................................................................................... 14
Download and Build Amazon FreeRTOS ......................................................................................................... 14
Getting Started with the Microchip Curiosity PIC32MZEF ............................................................................ 17
Setting Up Your Environment .......................................................................................................................... 17
Download and Build Amazon FreeRTOS ......................................................................................................... 17
Getting Started with the FreeRTOS Windows Simulator .................................................................................. 20
Setting Up Your Environment .......................................................................................................................... 20
Download and Build Amazon FreeRTOS ......................................................................................................... 20
Run the FreeRTOS Samples ........................................................................................................................... 23
Next Steps ...................................................................................................................................................... 25
Updating Firmware with OTA ............................................................................................................................ 25
OTA Update Prerequisites ................................................................................................................................ 25
Storing Your Firmware Image in Amazon S3 .................................................................................................... 29
Navigating the Example Project ....................................................................................................................... 30
Directory and File Organization ........................................................................................................................ 30
Configuration Files ......................................................................................................................................... 30
Thing Shadow Demo ......................................................................................................................................... 31
Greengrass Discovery Demo ............................................................................................................................ 32
OTA Demo .................................................................................................................................................... 33
Amazon FreeRTOS Developer Guide .................................................................................................................. 34
Amazon FreeRTOS Architecture .......................................................................................................................... 34
FreeRTOS Kernel Fundamentals ....................................................................................................................... 1
FreeRTOS Kernel Scheduler ........................................................................................................................... 35
Memory Management ....................................................................................................................................... 35
Inter-task Coordination ................................................................................................................................... 36
Software Timers ............................................................................................................................................... 39
Low Power Support ......................................................................................................................................... 39
FreeRTOS Libraries .......................................................................................................................................... 1
Cloud Connectivity .......................................................................................................................................... 40
Greengrass Connectivity .................................................................................................................................. 42
Amazon FreeRTOS Security ............................................................................................................................... 44
FreeRTOS Wi-Fi Interface ................................................................................................................................ 46
What Is Amazon FreeRTOS?

Amazon FreeRTOS is a microcontroller operating system based on the FreeRTOS kernel. It includes libraries for connectivity and security. You can use the Amazon FreeRTOS console to configure and download the FreeRTOS kernel and software libraries for your application. The Amazon FreeRTOS Qualification Program gives you the confidence that the microcontroller you choose fully supports the Amazon FreeRTOS features and capabilities.

The FreeRTOS Kernel

The FreeRTOS kernel is a real-time operating system 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.

Over-the-Air Updates (Beta)

Amazon FreeRTOS Over-the-Air (OTA) Updates is an end-to-end, cloud-based managed solution for secure over-the-air firmware updates on distributed fleets of small devices or microcontroller units (MCUs). An OTA update is the process of updating the firmware on remote devices. Internet-connected devices can be in use for a long time, and must be updated periodically to fix bugs and improve functionality. The OTA Updates service enables you to:

- Securely deploy new firmware images across a fleet of devices.
- Verify the authenticity and integrity of the new firmware.
- Deploy firmware to a small set of devices and test it before fleet-wide deployment.
- Monitor the progress of a deployment.
- Debug a failed deployment.
- Push deployments on devices as they are added to groups, factory-reset, or reprovisioned.

For more information about OTA updates, see:

- FreeRTOS Over-the-Air Updates (p. 52)
- Updating Firmware with OTA (p. 25)
- OTA Demo (p. 33)

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 Console

The Amazon FreeRTOS console is used to create software configurations that you can download onto your development computers to get started with Amazon FreeRTOS. The console allows you to select the hardware platform, compiler and IDE used for development, and Amazon FreeRTOS libraries required for your application. After you have made these selections, a custom zip file of the required files is created along with demo projects that make it easy to get started with Amazon FreeRTOS. This zip file includes the FreeRTOS kernel, Amazon FreeRTOS libraries, and board support packages (BSPs) and hardware drivers for the hardware platform.

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

During the development process, you can customize and download Amazon FreeRTOS source code from the Amazon FreeRTOS console. Each of your configurations is saved in AWS. You can download the source code using these configurations at any time. After you have the source code, you can 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 configure Amazon FreeRTOS and run it on one of the qualified microcontroller boards. In this tutorial, we assume you are familiar with the AWS IoT console. If not, we recommend that you start with the AWS IoT Getting Started tutorial.

Granting FreeRTOS Permissions

In order to access the Amazon FreeRTOS console, you must grant your IAM user AmazonFreeRTOSFullAccess permissions.

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.

Amazon FreeRTOS Prerequisites

Before you install Amazon FreeRTOS, you need an AWS account and one of the supported hardware platforms.

1. To create an AWS account, see Create and Activate an AWS Account.
2. Purchase one of the supported hardware platforms:
   • STMicroelectronicsSTM32L4 Discovery kit IoT node
   • Texas Instruments CC3220SF-LAUNCHXL
   • NXP LPC54018 IoT Module
   • Microchip Curiosity PIC32MZEF development kit
   • Microsoft Windows 7 or later, with at least a dual core and a hard-wired Ethernet connection

Create Your AWS IoT Credentials

A device connected to AWS IoT is represented by an IoT thing. The IoT thing is associated with a device certificate, private key, and an AWS IoT policy. The device uses the certificate and private key to authenticate with AWS IoT. The AWS IoT policy is associated with the certificate and determines which AWS IoT operations the device can perform. For more information, see Security and Identity in AWS IoT.

To create an AWS IoT policy

1. To create an IAM policy you will need to know your AWS region and AWS account number.
   
   To find your AWS account number, browse to the AWS console, select your username in the upper right-hand corner of the console, and 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 will 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. On the left-hand navigation pane choose Secure, Policies, and the Create button.
4. Type a name to identify your policy.
5. In the Add statements section choose Advanced mode. Replace `aws-region` and `aws-account` in the following JSON and then copy and paste the configuration into the policy editor window.

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

This policy grants the following permissions:

- `iot:Connect`
  
  Grants your device the permission to connect to the AWS IoT message broker.

- `iot:Publish`
  
  Grants your device the permission to publish an MQTT message on the `freertos/demos/echo` MQTT topic.
iot:Subscribe

Grants your device the permission to subscribe to the freertos/demos/echo MQTT topic filter.

iot:Receive

Grants your device the permission to receive message from the AWS IoT message broker.

6. Choose the **Create** button.

**To create an IoT thing 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**.
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 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 checkbox next to the policy you created previously.
11. Choose **Register Thing** to finish the thing registration process.

---

**Getting Started with the Texas Instruments (TI) CC3220SF-LAUNCHXL**

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

If you do not have the TI CC3220SF-LAUNCHXL development kit, you can purchase one from Texas Instruments. Use a USB cable to connect your Texas Instruments CC3220SF-LAUNCHXL to your computer.

**Setting Up Your Environment**

**Install TI Code Composer Studio**

2. Download the offline installer 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 installation issues, see TI Development Tools Support, Code Composer Studio FAQs, and Troubleshooting Code Composer Studio.

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.
4. Choose Download FreeRTOS Software.
5. Under Software Configurations, find Connect to AWS IoT- TI, and then choose Download.

6. Unzip the downloaded file to a folder, and make 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. To accommodate
the files in the Amazon FreeRTOS projects, make sure the path to the AmazonFreeRTOS
directory is less than 260 characters long. For example, C:\Users\Username\Dev
\AmazonFreeRTOS will work, but C:\Users\Username\Documents\Development
\Projects\AmazonFreeRTOS will cause build failures.

Import the Amazon FreeRTOS Sample Code into TI Code Composer

1. Open TI Code Composer and type a name for a new workspace.
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.
4. The project aws_demos should be selected by default.
5. To import the project into TI Code Composer, choose Finish.
6. From the Project menu, choose Build Project to make sure it compiles without errors or warnings.

Configure Your Project

Configure Your Wi-Fi Credentials

1. In the Project Explorer window, open aws_demos\application_code\common_demos\include
  \aws_clientcredential.h.
2. Specify values for the following #define constants:
   • clientcredentialWIFI_SSID: The SSID for your Wi-Fi network.
   • clientcredentialWIFI_PASSWORD: The password for your Wi-Fi network.
   • clientcredentialWIFI_SECURITY: The security type for your Wi-Fi network. Valid values are:
     • eWiFiSecurityOpen: Open, no security.
     • eWiFiSecurityWEP: WEP security.
     • eWiFiSecurityWPA: WPA security.
     • eWiFiSecurityWPA2: WPA2 security.

Configure Your AWS IoT Endpoint

You must specify a custom AWS IoT endpoint in order 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
  <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.
Configure Your AWS IoT Credentials

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

To format your certificate and private key

1. In a browser window, open `<BASE_FOLDER>\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under Certificate PEM file, choose the `<ID>-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the `<ID>-private.pem.key` you downloaded from the AWS IoT console.
4. Choose Generate and save aws_clientcredential_keys.h, and then save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the stub file that is 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. From the Run menu in TI Code Composer, choose Debug to start debugging.
6. When the debugger stops at the breakpoint in main(), go to the Run menu, and then choose Resume.

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

Troubleshooting

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

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 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 (ST) STM32L4 Discovery Kit IoT node

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

If you do not already have the STMicroelectronics STM32L4 Discovery kit IoT node, you can purchase one from STMicroelectronics. Use a USB cable to connect your STMicroelectronics STM32L4 Discovery kit IoT node to your computer.

**Setting Up Your Environment**

**Install System Workbench for STM32**

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

If you encounter installation issues, 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.
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 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. To accommodate the files in the Amazon FreeRTOS projects, make sure the path to the AmazonFreeRTOS directory is less than 260 characters long. For example, C:\Users\Username\Dev\AmazonFreeRTOS will work, but C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS will cause 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\stm32l475_discovery\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

#### Configure Your Wi-Fi Credentials

1. In the **Project Explorer** window, open `AmazonFreeRTOS\demos\common\include\aws_clientcredential.h`.

2. Specify values for the following `#define` constants:
   - `clientcredentialWIFI_SSID`: The SSID for your Wi-Fi network.
   - `clientcredentialWIFI_PASSWORD`: The password for your Wi-Fi network.
   - `clientcredentialWIFI_SECURITY`: The security type for your Wi-Fi network. Valid values are:
     - `eWiFiSecurityOpen`: Open, no security.
     - `eWiFiSecurityWEP`: WEP security.
     - `eWiFiSecurityWPA`: WPA security.
     - `eWiFiSecurityWPA2`: WPA2 security.

#### Configure Your AWS IoT Endpoint

You must specify a custom AWS IoT endpoint in order 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 `<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.

#### Configure Your AWS IoT Credentials

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

**To format your certificate and private key**

1. In a browser window, open `<BASE_FOLDER>\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under **Certificate PEM file**, choose the `<ID>-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under **Private Key PEM file**, choose the `<ID>-private.pem.key` you downloaded from the AWS IoT console.
4. Select the **Generate and save aws_clientcredential_keys.h** and save the file in `<BASE_FOLDER>\demos\common\include`. This overwrites the stub file that is 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. 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.
   6. 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 Amazon FreeRTOS 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

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, select Register with IAR Systems to get an evaluation license.

   Note
   If you encounter any installation issues, 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 Users’ 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.
4. Choose Download FreeRTOS Software.
5. In the Software Configurations, find Connect to AWS IoT- NXP, 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. To accommodate the files in the Amazon FreeRTOS projects, make sure the path to the AmazonFreeRTOS directory is less than 260 characters long. For example, C:\Users\Username\Dev\AmazonFreeRTOS will work, but C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS will cause build failures.

**Import the Amazon FreeRTOS Sample Code into IAR Embedded Workbench**

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.
Configure Your Project

Configure Your Wi-Fi Credentials

1. In the Project Explorer window, open \aws_demos\application_code\common_demos\include\aws_clientcredential.h.
2. Specify values for the following #define constants:
   - `clientcredentialWIFI_SSID`: The SSID for your Wi-Fi network.
   - `clientcredentialWIFI_PASSWORD`: The password for your Wi-Fi network.
   - `clientcredentialWIFI_SECURITY`: The security type for your Wi-Fi network. Valid values are:
     - `eWiFiSecurityOpen`: Open, no security.
     - `eWiFiSecurityWEP`: WEP security.
     - `eWiFiSecurityWPA`: WPA security.
     - `eWiFiSecurityWPA2`: WPA2 security.

Configure Your AWS IoT Endpoint

In order for the FreeRTOS sample code to connect to AWS IoT, you must specify your custom AWS IoT endpoint.

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 `<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.

Configure Your AWS IoT Credentials

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

**To format your certificate and private key**

1. In a browser window, open `<BASE_FOLDER>\demos\common\devmode_key_provisioning\CertificateConfigurationTool\CertificateConfigurator.html`.
2. Under Certificate PEM file, choose the `<ID>-certificate.pem.crt` you downloaded from the AWS IoT console.
3. Under Private Key PEM file, choose the `<ID>-private.pem.key` you downloaded from the AWS IoT console.
4. Select Generate and save aws_clientcredential_keys.h and save the file in `<BASE_FOLDER>\demos\common\include`. This will overwrite the stub file that is in the directory already.

Run the FreeRTOS Samples

In order 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 IAR Embedded Workbench, 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. In IAR Embedded Workbench, from the Project menu, choose Download and Debug.
8. From the Debug menu, choose Start Debugging.
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.

Note
If a J-Link ... Device Selection dialog box opens, choose OK to continue. In the Target Device Settings dialog box, choose Unspecified, choose Cortex-M4, and then choose OK. You only need to do this once.

Troubleshooting
If no messages are received at 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 Amazon FreeRTOS Prerequisites (p. 4).

If you do not have the Microchip Curiosity PIC32MZEF development kit, you can purchase one from Microchip. Use a USB cable to connect your Microchip Curiosity PIC32MZEF development kit to your computer.

Setting Up Your Environment

1. Install the latest Java SE SDK.
2. Install the latest version of the MPLAB X Integrated Development Environment.
3. Install the latest version of the MPLAB XC32 Compiler.
4. Connect the Ethernet module on the top of the base board to your computer.

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. To accommodate the files in the Amazon FreeRTOS projects, make sure the path to the AmazonFreeRTOS directory is less than 260 characters long. For example, **C:\Users\Username\Dev \AmazonFreeRTOS** will work, but **C:\Users\Username\Documents\Development \Projects\AmazonFreeRTOS** will cause build failures.

**Import the Amazon FreeRTOS Sample Code into 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`.
3. Choose **Open project** to import the project.
4. Press F11 or from the **Production** menu, choose **Build project**.

**Configure Your Project**

**Configure Your AWS IoT Endpoint**

In order for the FreeRTOS sample code to connect to AWS IoT, you must specify your custom AWS IoT endpoint.

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. (The endpoint 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.

**Configure Your AWS IoT Credentials**

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

**To format your certificate and private key**

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

**Run the FreeRTOS Samples**

In order to run the Amazon FreeRTOS demos on the Microchip Curiosity PIC32MZEF development kit, connect the USB port on the Microchip Curiosity PIC32MZEF to your host computer.

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 **Subscription topic**, type `freertos/demos/echo`, and then choose **Subscribe to topic**.
5. In the MPLAB IDE, from the **Project** menu, choose **Build**.
6. From the **Project** menu, choose **Download and Debug**.
7. From the **Debug** menu, choose **Start Debugging**.
8. When the debugger stops at the breakpoint in `main()`, from the **Debug** menu, 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 are received at the AWS IoT console, try the following:

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

---

**Getting Started with the FreeRTOS Windows Simulator**

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

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

**Setting Up Your Environment**

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

**Download and 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.
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 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. To accommodate the files in the Amazon FreeRTOS projects, make sure the path to the `AmazonFreeRTOS` directory is less than 260 characters long. For example, `C:\Users\Username\Dev\AmazonFreeRTOS` will work, but `C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` will cause 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 in order for the FreeRTOS sample code to connect to AWS IoT.

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

Configure Your AWS IoT Credentials

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

To format your certificate and private key

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

Output like the following should be displayed in the output window for the `aws_demo` application.

```
0 0 [None] FreeRTOS_IPInit
1 0 [None] vTaskStartScheduler

The following network interfaces are available:

Interface 1 - rpcap://\Device\NPF_{FD3E22FF-2E53-4791-8522-AD889B6ABF1F}
(Network adapter 'Intel(R) Ethernet Connection (3) I218-LM' on local host)

Interface 2 - rpcap://\Device\NPF_{260D089E-CECC-43A7-8FC3-021FA7D17903}
(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.
2 92 [IP-task] vDHCPProcess: offer 10.0.0.189
3 152 [IP-task] vDHCPProcess: offer 10.0.0.189
4 152 [IP-task] Creating MQTT Echo Task...
5 152 [IP-task]

IP Address: 10.0.0.189
6 152 [IP-task] Subnet Mask: 255.255.255.0
7 152 [IP-task] Gateway Address: 10.0.0.1
8 152 [IP-task] DNS Server Address: 75.75.75.75

9 152 [MQTT] Blocking on queue with timeout set to 0.
10 152 [MQTT] Timed out on queue read.
11 152 [MQTT] Blocking on queue with timeout set to ffffffff.
13 153 [MQTTEcho] Sending command to MQTT task.
14 153 [MQTT] Received message 10000 from queue.
15 212 [IP-task] Socket sending wakeup to MQTT task.
16 492 [MQTT] Blocking on queue with timeout set to 2ee0.
17 492 [MQTT] Received message 0 from queue.
18 492 [MQTT] Blocking on queue with timeout set to 2ee0.
19 552 [IP-task] Socket sending wakeup to MQTT task.
20 552 [MQTT] Received message 0 from queue.
21 552 [MQTT] MQTT Connect was accepted. Connection established.
22 552 [MQTT] Notifying task.
23 552 [MQTT] Blocking on queue with timeout set to 493e0.
24 552 [MQTTEcho] Command sent to MQTT task passed.
25 552 [MQTTEcho] MQTT echo connected.
26 552 [MQTTEcho] MQTT echo test echoing task created.
```
The sample sends 11 Hello World messages and then displays the following messages to indicate the sample has completed successfully:

```
350 61473 [MQTTEcho] Command sent to MQTT task passed.
351 61473 [MQTTEcho] MQTT echo demo finished.
```

In the AWS IoT console, the MQTT client displays the messages received from the FreeRTOS Windows simulator.
Next Steps

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
- Updating Firmware with OTA (p. 25)
- Navigating the Example Project (p. 30)
- Thing Shadow Demo (p. 31)
- Greengrass Discovery Demo (p. 32)
- OTA Demo (p. 33)

Updating Firmware with OTA

This section discusses how upload a firmware image to Amazon S3, create a stream, sign your firmware, and schedule an OTA update.

OTA Update Prerequisites

OTA Permissions

To schedule an OTA update in the AWS IoT console, you need an IAM user with the following permissions:

```json
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Effect":"Allow",
            "Action":[
                "s3:ListBucket",
                "s3:ListAllMyBuckets",
                "s3:CreateBucket",
                "s3:PutBucketVersioning",
                "s3:GetBucketLocation",
                "s3:GetObjectVersion",
                "acm:ImportCertificate",
                "acm:ListCertificates",
                "acm:ListCertificates",
                "iot:ListThings",
                "iot:ListThingGroups",
                "iot:CreateStream",
                "iot:CreateOTAUpdate",
                "iot:GetOTAUpdate",
                "iot:ListJobs",
                "iot:ListJobExecutionsForJob",
                "iot:DescribeJob",
                "iot:GetJobDocument",
                "iam:ListRoles",
                "signer:ListSigningJobs",
                "signer:StartSigningJob",
                "signer:DescribeSigningJob"
            ]
        }
    ]
}
```
OTA Update Prerequisites

For more information see Creating an IAM User, Creating IAM policies, and Attaching IAM policies.

**AWS IoT Things**

You must have an AWS IoT thing to represent each device you connect to AWS IoT. For more information about AWS IoT things, see Managing Things with AWS IoT. To create an AWS IoT thing, use the `create-thing` AWS CLI command:

```bash
aws iot create-thing --thing-name <my-thing-name>
```

The `create-thing` CLI command returns information about the thing:

```json
{
    "thingName": "my-thing-name",
    "thingArn": "arn:aws:iot:us-east-1:123456789012:thing/my-thing-name",
    "thingId": "68afbf50-ff2a-4298-975b-581ee02b7351"
}
```

You use the thing ARN when you create an OTA update deployment.

**AWS IoT Thing Groups**

You can create an AWS IoT thing group to organize your IoT things and schedule an OTA update on an AWS IoT thing group. The update will be scheduled for each thing associated with a thing group. For more information about AWS IoT thing groups, see Things Groups. To create an AWS IoT thing group, use the `create-thing-group` AWS CLI command:

```bash
$ aws iot create-thing-group --thing-group-name <group-name>
```

The `create-thing-group` CLI command returns information about the thing group:

```json
{
    "thingGroupName": "<group-name>",
    "thingGroupId": "abcdefgh12345678ijklmnop12345678qrstuvwx",
    "thingGroupArn": "arn:aws:iot:<your-aws-region>:<your-aws-account-id>:thinggroup/<group-name>"
}
```

You use the thing ARN when you create an OTA update deployment.
OTA IAM Role

The OTA Update service assumes an IAM role to create jobs on your behalf.

To create an OTA role:

2. From the navigation pane, choose Roles.
3. Choose Create role.
4. Under Select type of trusted entity, choose AWS Service.
5. Choose IoT from the list of AWS services.
6. Choose Next: Permissions.
7. Choose Next: Review.
8. Type a role name and description, and then choose Create role.
9. Type your role's name in the search box and choose your role from the list.
10. Choose the Attach policy button.
11. In the Search box, type AWSIoTOTAUpdate, choose it in the list of managed policies and choose the Attach policy button.
12. In the list of roles, choose your role again.
13. In the lower-right, choose Add inline policy.
14. Choose Custom Policy, and then choose Select.
15. Type a name for the inline policy and copy and paste the following policy document into the Policy Document text box, replacing <example-bucket> with your bucket's name.

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

Create and Import Certificate and Private Key

Create a file named cert_config with the following contents in your working directory.

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

[ my dn ]
    commonName = test_signer@amazon.com

[ my_exts ]
    keyUsage         = digitalSignature
    extendedKeyUsage = codeSigning
```
Create a new certificate and private key:

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

Import your certificate and private key into AWS Certificate Manager:

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

This command returns an ACM ARN. You use this value when you create an OTA update job.

**Code Signing for Amazon FreeRTOS**

You can use Code Signing for Amazon FreeRTOS to sign your code for any IoT device that is supported by Amazon Web Services (AWS). You upload your code image to an Amazon S3 bucket, import a code signing certificate into AWS Certificate Manager, and then call the `StartSigningJob` Code Signing API. Code Signing for Amazon FreeRTOS saves your signed code image to an Amazon S3 bucket that you specify. You can then distribute the signed code to devices. For more information, see the Code Signing for Amazon FreeRTOS Developer Guide and the Code Signing for Amazon FreeRTOS API Reference.

**Granting Access to the Code Signing Service**

In production environments you should digitally sign your firmware update to ensure the authenticity and integrity of the update. You can use the Code Signing for Amazon FreeRTOS for this purpose. You must grant access to the Code Signing for Amazon FreeRTOS service for your AWS account.

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

2. In the navigation pane, choose Policies.
3. Choose Create Policy.
4. Choose Create Your Own Policy.
5. Type a policy name and description.
6. For Policy Document, copy and paste the following JSON document. This policy allows access to all code signing operations to the IAM user with which it is associated.

   ```json
   {
       "Version": "2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Action": [
                   "signer:*"
               ],
               "Resource": [
                   "*"
               ]
           }
       ]
   }
   ```
7. In the navigation pane, choose Users.
8. Choose your IAM user account.
10. Choose Attach existing policies directly, and then choose the policy you just created.
Note
You must attach this policy document to the deployment role discussed later in this topic.

Signing Your Code with the AWS CLI

You can also use the AWS CLI to invoke Code Signing for Amazon FreeRTOS.

Start a new signing job using your ACM ARN. The following commands specify an Amazon S3 bucket version. Make sure that you enable bucket versioning when you upload your firmware to Amazon S3.

To sign for the Visual Studio simulator platform (RSA/SHA256):

```shell
```

To sign for the TI CC3220SF-LAUNCHXL platform (RSA/SHA1):

```shell
```

To update firmware on your devices you must create a new version of the firmware. The firmware image must include a version header at the application level which can be detected by the firmware performing the update. For more information, see the OTA Demo (p. 33). Additionally, the version must be higher than the version of firmware that performs the update or it will be rejected during the self test cycle. This is used to catch double resets and certain update failures.

When you have a firmware image created, upload it into an Amazon S3 bucket.

Storing Your Firmware Image in Amazon S3

To create an Amazon S3 bucket

2. Choose Create bucket.
3. Type a bucket name, and then choose Next.
4. On the Create bucket page, choose Versioning.
5. Choose Enable versioning, and then choose Save.
6. Choose Next.
7. Choose Create bucket.

To store your firmware image in Amazon S3

2. In the Search for buckets box, type your bucket name, and choose your bucket from the list.
3. Choose Upload.
4. Choose Add files, choose the firmware image you want to upload, and then choose Next.
5. Choose **Next**, and then choose **Next** again.
6. Choose **Upload**.

# Navigating the Example Project

## Directory and File Organization

There are two subfolders in the main Amazon FreeRTOS directory:

- **demos**
- **lib**

The **demos** directory 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 its own 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 `vStartpubsubDemotasks()` function is called. All others are commented. 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 the examples that can be executed by Amazon FreeRTOS can be found in the common directory under **demos**.

The **lib** directory contains the source code of the Amazon FreeRTOS libraries. The libraries that are available to you as part of Amazon FreeRTOS include:

- MQTT
- Thing 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 in order to create a version that runs on your platform. You can find configuration files for a given platform vendor 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.

Thing Shadow Demo

The thing shadow example demonstrates how to programmatically update and respond to changes in a thing shadow. The device in this scenario is a light bulb whose color can be set to red or green. The thing 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 thing 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 number of shadow update tasks that are created is controlled by the democonfigSHADOW_DEMO_NUM_TASKS constant defined in AmazonFreeRTOS/demos/common/shadow/aws_shadow_lightbulb_on_off.c.

prvShadowUpdateTasks generates an initial thing shadow document and updates the thing 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 (red -> green -> red).

prvUpdateTask responds to changes in the thing shadow's desired state. When the thing shadow's desired state changes, this task updates the thing shadow's reported state to reflect the new desired state.
1. Add the following policy to your device certificate:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:aws:iot:us-west-2:123456789012:client/<yourClientId>"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
        },
        {
            "Effect": "Allow",
            "Action": "iot:Receive",
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
        }
    ]
}
```

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

You can use the AWS IoT console to view your thing'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 thing shadows interact, see Thing Shadows Data Flow.

---

**Greengrass Discovery Demo**

Before you run the FreeRTOS Greengrass discovery demo, you must create a Greengrass group and 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 Create Your AWS IoT Credentials (p. 4).

After you have created an IoT thing for your Amazon FreeRTOS device, follow the instructions for setting up your environment and building Amazon FreeRTOS on one of the supported devices:

**Note**

Use the Getting Started 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).
Windows). Be sure to follow the steps in the "Configure Your Project" sections. Return to this topic after you have built Amazon FreeRTOS for your device.

- Getting Started with the Texas Instruments (TI) CC3220SF-LAUNCHXL (p. 6)
- Getting Started with the STMicroelectronics (ST) STM32L4 Discovery Kit IoT node (p. 10)
- Getting Started with the NXP LPC54018 IoT Module (p. 13)
- Getting Started with the Microchip Curiosity PIC32MZEF (p. 17)
- Getting Started with the FreeRTOS Windows Simulator (p. 20)

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

Amazon FreeRTOS includes a sample application that demonstrates the use of the OTA library. The OTA sample application is located in the demos\common\ota subdirectory. As a prerequisite to running the OTA sample, you must prepare and schedule an OTA update with AWS IoT as described in Updating Firmware with OTA (p. 25).

**Note**

The new firmware image being updated onto the device must include the OTA agent code (and be operational) in order to complete the update process. Failure to do this will leave OTA update jobs in an IN PROGRESS state.

The OTA sample app:

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.
Amazon FreeRTOS Developer Guide

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

Topics
- Amazon FreeRTOS Architecture (p. 34)
- FreeRTOS Kernel Fundamentals (p. 1)
- FreeRTOS Libraries (p. 1)
- Code Signing for Amazon FreeRTOS (p. 49)
- FreeRTOS Over-the-Air Updates (p. 52)
- Amazon FreeRTOS Console User Guide (p. 57)
- Downloading Amazon FreeRTOS from GitHub (p. 61)
- Amazon FreeRTOS Qualification Program (p. 61)
- Supported Platforms (p. 62)

Amazon FreeRTOS Architecture

Amazon FreeRTOS is intended for use on embedded microcontrollers, and is typically flashed to devices as a single compiled image with all the relevant 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 relevant drivers and BSP (board support packages) for the hardware platform. Independent of the individual microcontroller being used, embedded application developers can expect the same standardized interfaces to the FreeRTOS kernel and all Amazon FreeRTOS software libraries.

Amazon FreeRTOS Device Software Architecture

![Amazon FreeRTOS Device Software Architecture Diagram]
FreeRTOS Kernel Fundamentals

The FreeRTOS kernel is a real-time operating system that supports numerous architectures and 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 actually needs servicing. Blocked tasks do not require time-consuming periodic servicing. Direct-to-task notifications allow fast task signaling, with practically no RAM overhead, and 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, which allows you to provide an application-specific implementation appropriate for the real-time system being developed. 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 will be placed into the blocked state until either 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 behaviour 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 will be 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 will be 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 will receive 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 behaviour 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 will actually consume 14 bytes of buffer space. Likewise, writing a 100 byte message into a message buffer will actually use 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 will be 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, removes the task from the blocked state.

You can change this default behaviour 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 will be 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 behaviour 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:

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

Low Power Support

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

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

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

Topics
• Cloud Connectivity (p. 40)
• Greengrass Connectivity (p. 42)
• Amazon FreeRTOS Security (p. 44)
• FreeRTOS Wi-Fi Interface (p. 46)
• OTA Agent Library (p. 48)

Cloud Connectivity

Topics
• MQTT (p. 40)
• Thing Shadows (p. 41)

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 which 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 \texttt{pdTRUE} from the callback. You must then return the buffer back to the pool whenever you are done by calling \texttt{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 \texttt{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**

You can use these configuration options to customize the MQTT agent’s behavior:

- `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.

**Thing Shadows**

The Amazon FreeRTOS API provides functions to create, delete, and update a thing shadow. For more information, see [Thing Shadows](#). Thing shadows are accessed using the MQTT protocol. The FreeRTOS thing shadow API works with the MQTT API and handles the details of working with the MQTT protocol.

The Amazon FreeRTOS shadow APIs define functions to create, update, and delete thing shadows.

**Prerequisites**

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](#).
- `clientcredentialCLIENT_PRIVATE_KEY_PEM`: The private key PEM associated with your IoT thing. For more information, see [Configure Your AWS IoT Credentials](#).
Make sure the Amazon FreeRTOS MQTT library is installed on your device. For more information about the MQTT library, see MQTT (p. 40). Make sure that the MQTT buffers are large enough to contain the shadow JSON files. The maximum size for a thing shadow document is 8 KB. All default settings for the thing 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 thing shadow. For more information, see AWS IoT Getting Started.

Using the Thing 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 thing shadow:

- **SHADOW_Delete**
  - Delete the thing shadow.
- **SHADOW_Get**
  - Get the current thing shadow.
- **SHADOW_Update**
  - Update the thing shadow.

**Note**

When you are done working with the thing 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.
A Greengrass group and core device must be 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 because it 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 and manual selection. Automatic selection iterates through all 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` returns, 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. These last two functions use `GGD_SecureConnect_Read` to get the JSON data from the socket. `GGD_JSONRequestStart`, `GGD_SecureConnect_Send`, `GGD_JSONRequestGetSize` must be called in order to successfully 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 to which your FreeRTOS device can connect. To connect to a Greengrass core, call the `GGD_SecureConnect_Connect` function, passing in the core device's IP address, port, and certificate. To use manual selection, set the following fields of the `HostParameters_t` parameter:

```c
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.

```c
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 a variety of 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.

## 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 that sits 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. Please 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 work flow in order 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 only a small subset of the PKCS#11 interface standard to be implemented.

The following subset of PKCS#11 is used. This list is in roughly the order that the routines are called in support of provisioning, TLS client authentication, and clean-up. For detailed descriptions of the functions, please consult 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`

**Clean-up**

• `C_CloseSession`
• `C_Finalize`

**Asymmetric Cryptosystem Support**

The Amazon FreeRTOS PKCS#11 reference implementation supports 2048-bit RSA (signing only).

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aws --version</code></td>
<td><code>aws-cli/1.11.176 Python/2.7.9 Windows/8 botocore/1.7.34</code></td>
</tr>
<tr>
<td><code>openssl version</code></td>
<td><code>OpenSSL 1.0.2g 1 Mar 2016</code></td>
</tr>
</tbody>
</table>

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

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

7. Finally, run
   ```bash
   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 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 application 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 Wi-Fi on by initializing the radio, peripherals, and drivers. Your application must turn Wi-Fi on by calling `WIFI_On` before calling any other API. An application can turn Wi-Fi off by calling `WIFI_Off` to save power. This is useful for power constrained devices which can have intermittent connectivity. Calling `WIFI_Reset` will reset the Wi-Fi radio.

The Amazon FreeRTOS demos hard code Wi-Fi credentials into the application. If you are not able to 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 to 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`. 

46
A Wi-Fi device can be configured to be 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` will disconnect 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 which 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, a 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 support 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 upon the application and available power sources. A device may always be powered on to reduce latency or it may be intermittently connected and switch into a low power mode when Wi-Fi is not required. The interface API support 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 re-provision devices 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 limit of profiles you can save is platform dependent.

**Network Utilities**

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

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIFI_GetIP</td>
<td>Gets the IP address of a device.</td>
</tr>
<tr>
<td>WIFI_GetHostIP</td>
<td>Gets the host IP address.</td>
</tr>
</tbody>
</table>
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 keep firmware updates and the application running on your devices logically separated. 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.

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

1. Connect to the AWS IoT MQTT broker. This step is not required if you are not using AWS IoT device management, or if you prefer an alternative implementation of MQTT. For more information, see MQTT (p. 40).
2. Initialize the OTA agent by calling `OTA_AgentInit`. Your application must define an OTA callback function and a timeout. The callback implements application specific logic which will be executed after an OTA update job has completed. The timeout defines how long to wait for the initialization to complete.
3. If `OTA_AgentInit` timed out before the agent was ready, you may call `OTA_GetAgentState` to ensure that the agent is initialized and operating as expected.
4. When the OTA update is complete, your job completion callback is called and determines whether the update is accepted or rejected.
5. If the new firmware image has been rejected (for example, due to a validation error), the app can typically ignore the notification and just wait for the next update (or try again).
6. If the update is valid and has been marked as accepted, call `OTA_ActivateNewImage` in order to reset the device and boot the new firmware image.

Here is the complete OTA agent interface:

- `OTA_AgentInit`:
  Initializes the OTA engine. The caller provides messaging protocol callbacks and a timeout.
- `OTA_AgentShutdown`:
  Cleans up after using the OTA engine.
- `OTA_GetAgentState`:
  Gets the current state of the OTA agent.
- `OTA_ActivateNewImage`:
  Activates the newest microcontroller firmware image received through OTA. (The detailed job status should now be self-test.)
- `OTA_SetImageState`:
  Sets the validation state of the currently running microcontroller firmware image (testing, accepted or rejected).

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIFI_GetMAC</td>
<td>Gets the MAC address of a device.</td>
</tr>
<tr>
<td>WIFI_Ping</td>
<td>Sends a ping to a device on the network.</td>
</tr>
<tr>
<td>WIFI_Scan</td>
<td>Scans for available Wi-Fi networks.</td>
</tr>
</tbody>
</table>
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.

# Code Signing for Amazon FreeRTOS

You can use Code Signing for Amazon FreeRTOS to sign your code for any IoT device that is supported by Amazon Web Services (AWS).

Code Signing for Amazon FreeRTOS signs your code image and saves it in your destination S3 bucket. The signing certificate that you import into ACM can be self-signed or you can purchase a commercial code-signing certificate. You can use a self-signed certificate for testing. To use the Amazon FreeRTOS Windows Simulator, for example, you must have a self-signed RSA code signing certificate. For production, we recommend that you obtain a certificate from a well-known commercial certificate authority (CA). It is more likely that your customers will trust a certificate issued by a commercial 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 Extended Key Usage 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**

Code Signing for Amazon FreeRTOS is currently in beta. You can download the SDK from https://tools.signer.aws.a2z.com/awssigner-tools.zip.

The following steps outline the general procedure for using Code Signing for Amazon FreeRTOS:

1. Create source and destination Amazon S3 buckets.
2. Upload your unsigned code image to your source Amazon S3 bucket.
3. Import a code-signing certificate into ACM and retrieve its Amazon Resource Name (ARN).
4. Call the StartSigningJob function in the Code Signing API. Specify your certificate ARN and your source and destination buckets.

## Create a Self-Signed Certificate for the Amazon FreeRTOS Windows Simulator

The following example shows you how to use OpenSSL and the ACM AWS CLI to create and import a self-signed RSA certificate that you can use with the Amazon FreeRTOS Windows Simulator.

1. Create an OpenSSL configuration file. Name the file `cert_config` and copy the following configuration into it.

```bash
[ req ]
```
Initial Setup for OTA with Amazon FreeRTOS

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

In order for a firmware upgrade to be committed by the Amazon FreeRTOS OTA agent (installed on your device following the initial reboot), the firmware image needs to include the OTA agent library and be marked as an app version upgrade.

The first step for preparing an OTA is to build the image to install on your devices.

1. Open the aws_demos project in the appropriate IDE for your device.
2. Open aws_demo_runner.c and uncomment the call to vStartOTAUpdateDemoTask. Make sure all other function calls are commented out.
3. Open aws_ota_update_demo.c and set the value of xAppFirmwareVersion to your new firmware version. For example:

```c
const AppVersion32_t xAppFirmwareVersion = {
    .u.x.ucMajor = 0,
    .u.x.ucMinor = 9,
    .u.x.usBuild = 1,
};
```

4. Build the project and copy the aws_demo.bin file from the debug output directory of your project to another location. The next step will overwrite this file.
5. Change xAppFirmwareVersion back to its original value (0.9.0) and rebuild the project. This is the version you'll run in the debugger once you've got everything else ready to run.

OTA Setup for Texas Instruments CC3220SF-LAUNCHXL

The TI CC3220SF-LAUNCHXL supports two certificate chains for firmware code-signing: one for production ("certificate-catalog"), and one for testing and development ("certificate-playground"). To use the production certificate chain, you must purchase a commercial code-signing certificate and set the board to production mode using the TI Uniflash tool.
The playground certificate chain allows you to try out OTA updates without a commercial code-signing certificate. To use the playground certificate chain you must set the board to production mode using the TI Uniflash tool. Follow the steps below to create a new code signing certificate that is linked to the TI playground certificate hierarchy and meets AWS Certificate Manager and Code Signing for Amazon FreeRTOS criteria.

1. Install the SimpleLink CC3220 SDK version 1.40.01.00. By default, the files you need will be located in `C:\ti\simplelink_cc32xx_sdk_1_40_01_00\tools\cc32xx_tools\certificate-playground` on Windows and `/Applications/Ti/simplelink_cc32xx_sdk_1_40_01_00/tools/cc32xx_tools/certificate-playground` on macOS.

2. Create a new 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 TI playground root private key from DER format to PEM format.
   
   The TI playground root private key can be found in `C:\ti\simplelink_cc32xx_sdk_1_40_01_00\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert-key` on Windows or `/Applications/Ti/simplelink_cc32xx_sdk_1_40_01_00/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert-key` on macOS.

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

4. Convert the TI playground root CA certificate from DER format to PEM format.
   
   The TI playground root certificate can be found in `C:\ti\simplelink_cc32xx_sdk_1_40_01_00\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert` on Windows or `/Applications/Ti/simplelink_cc32xx_sdk_1_40_01_00/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert` on macOS.

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

5. Sign the CSR with the TI 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 will burn 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:
   
   ```
   ```

   **Note**
   This step assumes you are going to use Code Signing for Amazon FreeRTOS to sign your firmware images. Using Code Signing for Amazon FreeRTOS is recommended, but you can sign your firmware images manually if needed.
Configure the CC3220SF-LAUNCHXL to Trust your Certificate

1. On your TI developer board, place the SOP jumper on the middle set of pins (position = 1) and reset the board.
2. Attach the board to your computer using a USB cable.
3. Download, install, and run the TI Uniflash tool.
4. In Uniflash, under Detected Devices choose the Start Now button and then choose the Start button.
5. Choose New Project.
6. On the Start new project page, type a name for your project, and set Device Type to CC3220SF-LAUNCHXL, set Device Mode to Develop, and choose Create Project.
7. Choose the Connect button on the right-hand side of the Uniflash application window.
8. Under the Files heading, choose User Files on the left-hand side of the Uniflash application window.
9. In the File selector pane in the center of the window, choose the Add File icon.
10. Select dummy-root-ca-cert, choose Open, and then choose Write.
11. Use the same method to add tisigner.cert.der to your project.
12. In the Action drop-down list, choose Select MCU Image. Then choose Browse and choose the firmware image to use to upgrade your device (aws_demos.bin).
13. In the file dialog, confirm the file name is set to mcuflashimg.bin. Under the Private Key File Name drop-down, choose Browse and choose your code-signing private key (tisigner.key). Under Certification File Name, choose your code-signing certificate that you added to your image (tisigner.crt.der). Then choose Write button.
15. Un-check the Use default Trusted Root-Certificate Catalog checkbox.
16. Choose the Browse button under Source File and select simplelink_cc32xx_sdk_1_50_00_06\tools\cc32xx_tools\certificate-playground\certcatalogPlayGround20160911.lst.
17. Under Signature Source File choose the Browse button and select simplelink_cc32xx_sdk_1_50_00_06\tools\cc32xx_tools\certificate-playground\certcatalogPlayGround20160911.lst.signed.bin.
18. Choose the button on the right-hand side of the window.
19. Choose Choose Program Image (Create and Program).
20. After programming process is completed. Place the SOP jumper back onto the first set of pins (position = 0) and reset the board.

FreeRTOS Over-the-Air Updates

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

OTA Update includes the following components:

OTA Manager

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

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

Streaming Service

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

Code Signing for Amazon FreeRTOS

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

OTA Library and Agent

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

The OTA agent is responsible for downloading a new executable image from the cloud, validating the image in a way that is appropriate for the app and device, handling interruptions during the download, and managing updates that are separated into multiple sections. The OTA agent also supports a Greengrass-mediated OTA mode for devices that are not directly connected to the cloud. In the Greengrass-mediated OTA mode, the update is downloaded by a trusted Greengrass core, which then pushes the update to Amazon FreeRTOS devices connected to it.

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

Device Startup

When a device starts up, it should do the following:

1. Subscribe to the device's notify topic: $aws/things/thing-name/jobs/notify.
2. Request pending job executions by publishing an empty message to: $aws/things/thingName/jobs/get.

For each device specified in a job, the OTA Update service publishes a message on the device's notify topic. The message looks like the following:

```json
{
  "timestamp": 1476214217017,
  "jobs": {
    "QUEUED": [
      {"jobId": "1111-2222-3333-4444",
       "queuedAt": 1476214216981,
       "lastUpdatedAt": 1476214216981
      }]
  }
}
```
Responding to an Update

When a device receives a notify message, it can retrieve the job document that describes the update by publishing a message to $aws/things/thing-name/jobs/job-id/get. If the request is successful, the AWS IoT Jobs service responds by publishing a message on $aws/things/thing-name/jobs/job-id/get/accepted:

```json
{
  "ts_ota": {
    "files": [
      {
        "streamname": "<your-stream-name>",
        "filepath": "<sample/path>",
        "version": null,
        "filesize": 100000,
        "fileid": 0,
        "attr": 0,
        "certfile": "certificate.pem",
        "sig-rsa-sha256": "<your-signature>
      }
    ]
  }
}
```

**Note**

If the request fails, the AWS IoT Jobs service publishes a message on $aws/things/thing-name/jobs/job-id/get/rejected.

Retrieving an Update

The device now has the job document, which it can interpret to perform the update for the job. The device fetches the stream name from the job document and subscribes to the streaming topic: $aws/things/thing-name/streams/stream-name. The device requests chunks of the update by publishing a message on $aws/things/thing-name/streams/stream-name/get.

**Note**

If the request fails, the Streaming service publishes a message on $aws/things/thing-name/streams/stream-name/get/rejected.

Reporting Update Progress

As the device is performing the update, it can report the status of the update by publishing messages on $aws/things/thing-name/jobs/job-id/update:

```json
{
  "status":"IN_PROGRESS",
  "statusDetails": {
    "progress":"50%"
  },
  "expectedVersion":"1",
  "returnValues":"NONE",
  "clientToken":"client001"
}
```
The AWS IoT Jobs service responds on `$aws/things/thing-name/jobs/job-id/update` accepted or `$aws/things/thing-name/jobs/job-id/update/rejected`. For example:

```json
{
    "clientToken":"client001",
    "timestamp":1476289222841
}
```

When the device completes the update, it publishes a message on `$aws/things/thing-name/jobs/job-id/update`, setting the status to `SUCCEEDED` if the update succeeded. For example:

```json
{
    "status":"SUCCEEDED",
    "statusDetails": {
        "progress":"100%"
    },
    "expectedVersion":"2",
    "returnValues":"NONE",
    "clientToken":"client-001"
}
```

If the update failed, the device publishes a message on `$aws/things/thing-name/jobs/job-id/update`, setting the status to `FAILED`. For example:

```json
{
    "status":"FAILED",
    "statusDetails": {
        "errorCode":"101",
        "errorMsg":"Unable to install update"
    },
    "expectedVersion":"2",
    "returnValues":"NONE",
    "clientToken":"client-001"
}
```

When the AWS IoT Jobs service receives this message, it publishes a message on the `$aws/things/thing-name/jobs/notify` topic to indicate the job execution is complete:

```json
{
    "timestamp":1476290692776,
    "jobs":{}
}
```

If there are other update job executions pending for the device, they are included in the message published to `$aws/things/thing-name/jobs/notify`. For example:

```json
{
    "timestamp":1476290692776,
    "jobs":{
        "QUEUED":[]{
            "jobId":"0002",
            "queuedAt":1476290646230,
            "lastUpdatedAt":1476290646230
        }
    },
    "IN_PROGRESS":[]{
        "jobId":"0003",
        "queuedAt":1476290646230,
```
OTA Security

The following are three aspects of security for Amazon FreeRTOS OTA:

Connection security

OTA Update relies on existing security mechanisms like TLS mutual authentication, used by AWS IoT. Traffic for OTA Update goes 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 the firmware

To ensure that the firmware is from a reliable source and has not been tampered with, you can use Code Signing for Amazon FreeRTOS to sign the 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 will undergo standard IAM Sigv4 authentication and authorization. To create a deployment you must have permissions to invoke the CreateDeployment, CreateJob and CreateStream API. In addition, in your Amazon S3 bucket policy/ACL, you must give read permissions to AWS IoT service principal so that we can access the firmware update stored in Amazon S3 during streaming.

Monitoring Updates

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

DescribeJobExecution

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

ListJobExecutionsForJob

Lists all job executions for a job.

ListJobExecutionsForThing

Get the list of all job executions for a thing.

For more information, see AWS IoT Jobs API

OTA Workflow

1. As a one-time bootstrapping step, you create or purchase a code-signing certificate and a private key and import these assets into the AWS Certificate Manager (ACM).

2. You deploy a device with factory provisioned firmware (for example, v1.0). The v1.0 firmware is configured to trust the code-signing certificate created in step 1.
3. When a firmware update is required, you make the required code changes and build the new firmware image.

4. You upload the new firmware image into an Amazon S3 bucket.

5. You digitally sign the new firmware image. You may either do this step manually, or use AWS Code Signing for Amazon FreeRTOS. This procedure makes use of the private key and digital certificate previously imported into ACM.

6. Using the AWS IoT Device Management console, create a data stream based on the digitally signed image produced by the previous step.

7. Using the AWS IoT Device Management console, schedule an OTA update to target a selected set of devices in the AWS IoT registry. When ready, start the update. The update streams the OTA image in chunks, over MQTT, to each selected device.

8. The Amazon FreeRTOS OTA agent on the device receives the chunked stream and reassembles the image.

9. Depending on the portable characteristics of the OTA agent and the device bootloader, the digital signature, checksum, and version number of the new image are verified.

10. Reset the board and, based on application-defined logic, commit the update. This includes notifying AWS IoT Device Management that the device has (or has not) successfully completed the OTA update.

Amazon FreeRTOS Console User Guide

Managing Amazon FreeRTOS Configurations

You can use the AWS IoT Device Management Console to manage software configurations and download Amazon FreeRTOS software for your device. The Amazon FreeRTOS software is prequalified on a variety of platforms. It includes the required hardware drivers, libraries, and example projects to help get you started quickly. You can choose between predefined configurations or create custom configurations.

Predefined configurations are defined for the prequalified platforms:

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

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

Custom configurations allow you to specify your hardware platform, integrated development platform (IDE), compiler, and only those RTOS libraries you require. This leaves more space on your devices for application code.

To create a custom configuration

1. Browse to the Amazon FreeRTOS console and choose Create new.
2. On the New Software Configuration page, choose Select a hardware platform, and choose one of the prequalified platforms.
3. Choose the IDE and compiler you want use.
4. For the Amazon FreeRTOS libraries you require, choose Add Library. If you choose a library that requires another library, it is added for you. If you want to choose more libraries, choose Add another library.
5. In the Demo Projects section, enable one of the demo projects. This enables the demo in the project files.
6. In Name required, type a name for your custom configuration.
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.

OTA Updates

You can use the AWS IoT device management console to manage OTA updates. Before going through this section, make sure you have completed the Getting Started with Amazon FreeRTOS (p. 4) section.

Attach a Policy to your IAM User

Attach the following policy to your IAM user.

```json
{
   "Version":"2012-10-17",
   "Statement":[
   {
      "Effect":"Allow",
      "Action":[
      "s3:ListBucket",
      "s3:ListAllMyBuckets",
      "s3:CreateBucket",
      "s3:PutBucketVersioning",
      "s3:GetBucketLocation",
      "s3:GetObjectVersion",
      "acm:ImportCertificate",
      "acm:ListCertificates",
      "iot:ListThings",
      "iot:ListThingGroups",
      "iot:CreateStream",
      "iot:CreateOTAUpdate",
      "iot:GetOTAUpdate",
      "iot:ListJobs",
      "iot:ListJobExecutionsForJob",
      "iot:DescribeJob",
      "iot:GetJobDocument",
      "iam:ListRoles",
      "signer:ListSigningJobs",
      "signer:StartSigningJob",
      "signer:DescribeSigningJob"
   ],
   "Resource":"*"
   },
   {
      "Effect":"Allow",
      "Action":[
      "s3:GetObject",
      "s3:PutObject"
   ],
   "Resource":"arn:aws:s3:::<example-bucket>/*"
   },
   {
      "Effect":"Allow",
      "Action":[
      "s3:GetObject",
      "s3:PutObject"
   ],
   "Resource":"arn:aws:s3:::<example-bucket>/*"
   }
}
```
“Action”: “iam:PassRole”,
“Resource”: “arn:aws:iam::<your-account-id>:role/<role-name>”
}
]
]

For more information about creating an IAM policy, see Creating IAM Policies. For more information about attaching a policy to an IAM user, see Attaching IAM Policies.

Create an OTA Update Service Role

The OTA Update service assumes an IAM role to create jobs on your behalf.

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. Choose Next: Permissions.
7. Choose Next: Review.
8. Type a role name and description, and then choose Create role.
9. Type your role’s name in the search box and choose your role from the list.
10. Choose the Attach policy button.
11. In the Search box, type AWSIoTOTAUpdate, choose it in the list of managed policies and choose the Attach policy button.
12. In the list of roles, choose your role again.
13. In the lower-right, choose Add inline policy.
14. Choose Custom Policy, and then choose Select.
15. Type a name for the inline policy and copy and paste the following policy document into the Policy Document text box, replacing <example-bucket> with your bucket’s name.

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

To create an OTA update

Important
Before following these instructions make sure you have created an S3 bucket with versioning enabled. For more information see, Storing Your Firmware Image in Amazon S3 (p. 29).

1. From the AWS IoT console, in the navigation pane, choose Manage and then Jobs.
2. If this is your first time creating an OTA update, select Create a Job, otherwise, in the upper right-hand corner of the console choose Create.
3. Choose Create OTA update job.
4. Under Select devices to update, choose the things or thing groups that represent the devices you want to update.
5. Under Select and sign your firmware image, choose Sign a new firmware image for me.
6. Under Pathname of code signing certificate on device, type the file path of the code signing certificate on your device.
   
   **Note**
   This code signing certificate is different from the device certificate used to authenticate your device.

7. Under Pathname of firmware image on device, type the file path of the firmware image on your device. This is the location where the firmware image is copied during the OTA update. Please check device specific instructions for any requirements for this field.
   
   **Note**
   If the pre-populated firmware pathname disappears, please refresh the page and it should be repopulated.

8. Under Device hardware platform, choose your hardware platform.
9. Under Select your firmware image in Amazon S3, choose your firmware image. The file must be a .bin file. If this is your first time creating an image:
   
   1. Choose the Create S3 Bucket button.
   2. Type a name for your bucket.
   3. Choose Upload an image and navigate to the .bin file that contains your firmware image.

   If this is your first time uploading credentials:
   
   1. Choose Import a certificate.
   2. Copy and paste the following files in PEM format into the appropriate text boxes
      
      • The code signing certificate.
      • The code signing private key.
      • The root of trust chain for the code signing certificate. If the certificate is self-signed this field can be left empty.
   3. Choose Import.
   4. If you are managing multiple certificate sets, remember the certificate ID so you will be able to identify it later on.
   5. Under Job type choose the appropriate job type. A snapshot job will deploy the update to the specified devices and then terminate. A continuous job will deploy the update to a specified thing group and will not terminate. When new things are added to the thing group, they will be updated automatically
   6. Under IAM role for OTA update select a role that grants AWS IoT access to your Amazon S3 bucket and images, AWS IoT Jobs, and AWS Code Signing for Amazon FreeRTOS.
   7. Type an ID and a description for your OTA update.
   8. Choose Create to create the OTA update.

If you choose Manage and Jobs in the navigation pane, you should see that a job was created. If you choose the job you can see more information about the status of the job. If your thing is already provisioned with credentials and running the OTA demo, the update will automatically begin. Otherwise, follow the instructions for getting your device set up for OTA. OTA Demo (p. 33).
10. Under **Code signing certificate**, choose **Import a certificate** to import a new certificate or choose **Select** to use an existing certificate.

11. Under **Job type**, choose the appropriate job type. A snapshot job deploys the update to the specified devices and then terminates. A continuous job deploys the update to a specified thing group and does not terminate. When new things are added to the thing group, they are updated automatically.

12. Under **IAM role for OTA update job**, choose a role that grants AWS IoT access to your Amazon S3 bucket and images, AWS IoT Jobs, and AWS Code Signing for Amazon FreeRTOS.

13. Type an ID and a description for your OTA update.

14. Choose **Create** to create the OTA update.

---

**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 interoperable 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 the relevant 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 send an acknowledgement and support you through the qualification steps.
Sign Up for the AWS Partner Network

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

Jointly Agree on Terms and Conditions

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

Pass Qualification Test Suite

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

The high-level steps are:

1. Download the latest version of the Amazon FreeRTOS source code.
2. Create a project using the target IDE and compiler that demonstrates the equivalent of a “Hello World” example for the target MCU.
3. Add Amazon FreeRTOS files and resources to the created project, and confirm the project still builds. At this stage, the included TQP tests should build and run, but are expected to fail because they have not yet been ported to your hardware.
4. Enable each Amazon FreeRTOS feature to work successfully on your MCU. This involves implementing each hardware-dependent layer of the Amazon FreeRTOS feature abstractions. Test these implementations and fix issues.
5. When all included tests are passing, 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, 1MB Flash, 256KB 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, and as such, contains limitations on its use. We recommend only using the Secure Sockets APIs from low priority tasks, and to limit transmit throughput. Future revisions are planned to improve this interface. For more information about this platform, see STMicroelectronics STM32L4 Discovery 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 360KB of SRAM, 128Mb 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 200MHz 32-bit MIPS M-class core with 2MB of Flash and 512KB of SRAM. For users who need to use Ethernet, the LAN8720A Ethernet PHY daughter board can be connected to the Curiosity PIC32MZEF development board. For more information about this platform, see Microchip PIC32MZ2048EFM100.
Amazon FreeRTOS Porting Guide

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

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

Topics
- Logging (p. 64)
- Connectivity (p. 65)
- Security (p. 67)
- Using Custom Libraries with Amazon FreeRTOS (p. 69)
- OTA Portable Abstraction Layer (p. 69)

Logging

Amazon FreeRTOS provides a thread-safe logging task that can be used by calling the configPRINTF function. configPRINTF is designed to behave like printf. To port configPRINTF, initialize your communications peripheral, and define the configPRINT_STRING macro so that it takes an input string and displays it on your preferred output.

Logging Configuration

configPRINT_STRING should be defined for your board's implementation of logging. Current examples use a UART serial terminal, but other interfaces can also be used.

```
#define configPRINT_STRING( x )
```

Use configLOGGING_MAX_MESSAGE_LENGTH to set the maximum number of bytes to be printed. Messages longer than this length are truncated.

```
#define configLOGGING_MAX_MESSAGE_LENGTH
```

When configLOGGING_INCLUDE_TIME_AND_TASK_NAME is set to 1, all printed messages are prepended with additional debug information (the message number, FreeRTOS tick count, and task name).

```
#define configLOGGING_INCLUDE_TIME_AND_TASK_NAME    1
```

vLoggingPrintf is the name of the FreeRTOS thread-safe printf call. You do not need to change this value to use AmazonFreeRTOS logging.

```
#define configPRINTF( X )    vLoggingPrintf X
```
**Connectivity**

You must first configure your connectivity peripheral. You can use Wi-Fi, Bluetooth, Ethernet, or other connectivity mediums. At this time, only a Wi-Fi management API is defined for board ports, but if you are using Ethernet, the FreeRTOS TCP/IP software can provide a good place to start.

### Wi-Fi Management

The Wi-Fi management library supports network connectivity following the 802.11 (a/b/n) protocol. If your hardware does not support Wi-Fi, you do not need to port this library.

The functions that must be ported are listed in the `lib/wifi/portable/<vendor>/<platform>/aws_wifi.c` file. You can find a detailed explanation for each public interface in `lib/include/aws_wifi.h`.

The following functions must be ported:

```c
WiFiReturnCode_t WIFI_On( void );
WiFiReturnCode_t WIFI_Off( void );
WiFiReturnCode_t WIFI_ConnectAP( const WiFiNetworkParams_t * const pxNetworkParams );
WiFiReturnCode_t WIFI_Disconnect( void );
WiFiReturnCode_t WIFI_Reset( void );
WiFiReturnCode_t WIFI_Scan( WiFiScanResult_t * pxBuffer, uint8_t uxNumNetworks );
```

### Sockets

The sockets library supports TCP/IP network communication between your board and another node in the network. The sockets APIs are based on the Berkeley sockets interface, but also include a secure communication option. At this time, only client APIs are supported. We recommend that you port the TCP/IP functionality first, before you add the TLS functionality.

Libraries for MQTT, Shadow, and Greengrass all make calls into the sockets layer. A successful port of the sockets layer allows the protocols built on sockets to just work.

### Major Differences from Berkeley Sockets Implementation

#### Security

The sockets interface must be configured to use TLS for secure communication. The `SetSockOpt` command has a couple of nonstandard options that must be implemented to work with AmazonFreeRTOS examples.

```c
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. 44). For information about porting TLS and cryptographic operations, see the TLS (p. 44) and Public Key Cryptography Standard #11 (p. 44) 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.

**Secure Sockets Porting for the STM32 IoT Discovery Kit**

- 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.

**Secure Sockets Porting for the TI CC3220SF-LAUNCHXL Board**

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 receiving 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.

---

**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)
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.

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