
FreeRTOS

Porting Guide



FreeRTOS: Porting Guide

Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

FreeRTOS Porting	1
What is FreeRTOS?	1
Porting FreeRTOS	2
System Requirements	2
Porting Older Versions of FreeRTOS	2
Porting FAQs	6
Downloading FreeRTOS for Porting	8
Setting Up Your FreeRTOS Source Code for Porting	9
Configuring the FreeRTOS Download	9
Configuring Directories for Vendor-supplied, Board-specific Libraries	9
Configuring Directories for Project Files	10
Configuring FreeRTOSConfig.h	10
Setting Up Your FreeRTOS Source Code for Testing	10
Creating an IDE Project	11
Creating a CMake List File	12
Porting the FreeRTOS Libraries	25
Porting Flowchart	25
configPRINTF_STRING()	27
Prerequisites	27
Implementation	27
Testing	27
FreeRTOS Kernel	28
Prerequisites	28
Configuring the FreeRTOS Kernel	28
Testing	29
Wi-Fi	29
Prerequisites	29
Porting	30
Testing	30
Validation	34
TCP/IP	35
Porting FreeRTOS+TCP	35
Porting lwIP	38
Secure Sockets	39
Prerequisites	40
Porting	40
Testing	40
Setting Up an Echo Server	43
Validation	45
PKCS #11	45
Prerequisites	45
Porting	45
Testing	47
Validation	49
TLS	49
Prerequisites	49
Porting	50
Connecting Your Device to AWS IoT	50
Setting Up Certificates and Keys for the TLS Tests	52
Creating a BYOC (ECDSA)	57
Testing	65
Validation	67
MQTT	67
Prerequisites	67

Setting Up the IDE Test Project	67
Configuring the CMakeLists.txt File	67
Setting Up Your Local Testing Environment	68
Running the Tests	68
Validation	68
HTTPS	68
Prerequisites	68
Setting Up the IDE Test Project	69
Configuring the CMakeLists.txt File	69
Setting Up Your Local Testing Environment	69
Running the Tests	69
Over-the-Air (OTA) Updates	69
Prerequisites	70
Porting	70
IoT Device Bootloader	71
Testing	74
Validation	76
Bluetooth Low Energy	76
Prerequisites	76
Porting	77
Testing	79
Validation	80
Common I/O	80
Prerequisites	81
Testing	81
Porting the I2C Library	83
Porting the UART Library	86
Porting the SPI Library	88
Migrating from Version 1.4.x to Version 201906.00 (and newer)	90
Migrating Applications	90
Migrating Ports	90
FreeRTOS Code Directory Structure	90
CMake Build System	90
Migrating the Wi-Fi Library Port	91

FreeRTOS Porting

What is FreeRTOS?

Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 175 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

FreeRTOS includes libraries for connectivity, security, and over-the-air (OTA) updates. FreeRTOS also includes demo applications that show FreeRTOS features on qualified boards.

FreeRTOS is an open-source project. You can download the source code, contribute changes or enhancements, or report issues on the GitHub site at <https://github.com/aws/amazon-freertos>. We release FreeRTOS code under the MIT open source license, so you can use it in commercial and personal projects.

We also welcome contributions to the FreeRTOS documentation (*FreeRTOS User Guide*, *FreeRTOS Porting Guide*, and *FreeRTOS Qualification Guide*). The markdown source for the documentation is available at <https://github.com/awsdocs/aws-freertos-docs>. It is released under the Creative Commons (CC BY-ND) license.

The FreeRTOS kernel and components are released individually and use semantic versioning. Integrated FreeRTOS releases are made periodically. The three types of FreeRTOS releases are major, minor, and long-term support (LTS). A major denotation indicates the addition of new features or significant updates to multiple libraries. All releases use date-based versioning with the format YYYYMM.NN, where:

- Y represents the year.
- M represents the month.
- N represents the release order within the designated month (00 being the first release).

For example, a second release in June 2019 would be 201906.01.

Previously, FreeRTOS releases used semantic versioning for major releases. Although it has moved to date-based versioning (FreeRTOS 1.4.8 updated to FreeRTOS 201906.00), the FreeRTOS kernel and each individual FreeRTOS library still retains semantic versioning. In semantic versioning, the version number itself (X.Y.Z) indicates whether the release is a major, minor, or point release. This can create situations where semantic versioning indicates a major release based on changes that don't affect an individual application. You can use the semantic version of a library to assess the scope and impact of a new release on your application.

LTS releases are maintained differently than other release types. Major and minor releases are frequently updated with new features in addition to defect resolutions. LTS releases are only updated with changes to address critical defects and security vulnerabilities. No new features are introduced in a given LTS release after launch. They are maintained for at least three calendar years after release, and provide device manufacturers the option to use a stable baseline as opposed to a more dynamic baseline represented by major and minor releases.

Porting FreeRTOS to Your IoT Device

Before a microcontroller board can run FreeRTOS, some FreeRTOS code must be ported to the device's hardware. Basic kernel ports should refer to the [FreeRTOS porting guide](#). For ports intending to include the FreeRTOS libraries for security, connectivity, etc., the following instructions build on the kernel port.

To port FreeRTOS to your device

1. Follow the instructions in [Downloading FreeRTOS for Porting \(p. 8\)](#) to download the latest version of FreeRTOS for porting.
2. Follow the instructions in [Setting Up Your FreeRTOS Source Code for Porting \(p. 9\)](#) to configure the files and folders in your FreeRTOS download for porting and testing.
3. Follow the instructions in [Porting the FreeRTOS Libraries \(p. 25\)](#) to port the FreeRTOS libraries to your device. Each porting topic includes instructions on testing the ports.

System Requirements

The device that you port to FreeRTOS must be a microcontroller board that meets the following minimum requirements:

- 25MHz processing speed
- 64KB RAM
- 128KB program memory per executable image stored on the MCU
- (If [Porting the OTA Library \(p. 69\)](#)) Two executable images stored on the MCU

Porting Older Versions of FreeRTOS

If you are porting an older version of FreeRTOS, go to the [amazon-freertos GitHub repository](#), and checkout the version of FreeRTOS that you are porting by its version tag. The qualification and testing documentation will be in PDF format, in the [tests](#) folder. See the table below for the qualification and testing documentation history.

Revision History of FreeRTOS Porting and Qualification Documentation

Date	Porting and Qualification Documentation Version	Change History	FreeRTOS Version
February 17, 2020	202002.00 (Porting Guide) 202002.00 (Qualification Guide)	<ul style="list-style-type: none">• Release 202002.00• "Amazon FreeRTOS..." is now "FreeRTOS..."	202002.00
December 17, 2019	201912.00 (Porting Guide) 201912.00 (Qualification Guide)	<ul style="list-style-type: none">• Release 201912.00• Added Porting the Common I/O Libraries (p. 80).	201912.00
October 29, 2019	201910.00 (Porting Guide)	<ul style="list-style-type: none">• Release 201910.00	201910.00

Date	Porting and Qualification Documentation Version	Change History	FreeRTOS Version
	201910.00 (Qualification Guide)	<ul style="list-style-type: none"> Updated random number generator porting information. 	
August 26, 2019	201908.00 (Porting Guide) 201908.00 (Qualification Guide)	<ul style="list-style-type: none"> Release 201908.00 Added Configuring the HTTPS Client Library for Testing (p. 68) Updated Porting the PKCS #11 Library (p. 45) 	201908.00
June 17, 2019	201906.00 (Porting Guide) 201906.00 (Qualification Guide)	<ul style="list-style-type: none"> Release 201906.00 Directory structured updated 	201906.00 Major
May 21, 2019	1.4.8 (Porting Guide) 1.4.8 (Qualification Guide)	<ul style="list-style-type: none"> Porting documentation moved to the FreeRTOS Porting Guide Qualification documentation moved to the FreeRTOS Qualification Guide 	1.4.8
February 25, 2019	1.1.6	<ul style="list-style-type: none"> Removed download and configuration instructions from Getting Started Guide Template Appendix (page 84) 	1.4.5 1.4.6 1.4.7
December 27, 2018	1.1.5	<ul style="list-style-type: none"> Updated Checklist for Qualification appendix with CMake requirement (page 70) 	1.4.5 1.4.6
December 12, 2018	1.1.4	<ul style="list-style-type: none"> Added lwIP porting instructions to TCP/IP porting appendix (page 31) 	1.4.5

Date	Porting and Qualification Documentation Version	Change History	FreeRTOS Version
November 26, 2018	1.1.3	<ul style="list-style-type: none">• Added Bluetooth Low Energy porting appendix (page 52)• Added AWS IoT Device Tester for FreeRTOS testing information throughout document• Added CMake link to Information for listing on the FreeRTOS Console appendix (page 85)	1.4.4
November 7, 2018	1.1.2	<ul style="list-style-type: none">• Updated PKCS #11 PAL interface porting instructions in PKCS #11 porting appendix (page 38)• Updated path to <code>CertificateConfigurator.html</code> (page 76)• Updated Getting Started Guide Template appendix (page 80)	1.4.3

Date	Porting and Qualification Documentation Version	Change History	FreeRTOS Version
October 8, 2018	1.1.1	<ul style="list-style-type: none">• Added new "Required for AFQP" column to <code>aws_test_runner_config.h</code> test configuration table (page 16)• Updated Unity module directory path in Create the Test Project section (page 14)• Updated "Recommended Porting Order" chart (page 22)• Updated client certificate and key variable names in TLS appendix, Test Setup (page 40)• File paths changed in Secure Sockets porting appendix, Test Setup (page 34); TLS porting appendix, Test Setup (page 40); and TLS Server Setup appendix (page 57)	1.4.2
August 27, 2018	1.1.0	<ul style="list-style-type: none">• Added OTA Updates porting appendix (page 47)• Added Bootloader porting appendix (page 51)	1.4.0 1.4.1

Date	Porting and Qualification Documentation Version	Change History	FreeRTOS Version
August 9, 2018	1.0.1	<ul style="list-style-type: none">• Updated "Recommended Porting Order" chart (page 22)• Updated PKCS #11 porting appendix (page 36)• File paths changed in TLS porting appendix, Test Setup (page 40), and TLS Server Setup appendix, step 9 (page 51)• Fixed hyperlinks in MQTT porting appendix, Prerequisites (page 45)• Added AWS CLI config instructions to examples in Instructions to Create a BYOC appendix (page 57)	1.3.1 1.3.2
July 31, 2018	1.0.0	Initial version of the FreeRTOS Qualification Program Guide	1.3.0

Porting FAQs

What is a FreeRTOS port?

A FreeRTOS port is a board-specific implementation of APIs for the required FreeRTOS libraries and the FreeRTOS that your platform supports. The port enables the APIs to work on the board, and implements the required integration with the device drivers and BSPs that are provided by the platform vendor. Your port should also include any configuration adjustments (e.g. clock rate, stack size, heap size) that are required by the board.

My device does not support Wi-Fi, Bluetooth Low Energy, or over-the-air (OTA) updates. Are all libraries required to port FreeRTOS?

The primary requirement for porting FreeRTOS connectivity libraries is that your device can connect to the cloud. If, for example, you can connect to the cloud across a secure ethernet connection, FreeRTOS, the Wi-Fi library is not a required. Keep in mind that some test and demo applications will not work without all of the libraries ported.

Can I reach an "echo server" from two different networks (for example, from two subnets across 2 different access points)?

An echo server is required to pass the TCP/IP and TLS port tests. The echo server must be reachable from the network that a board is connected to. Please consult your IT support to enable routing across subnets if you need devices on different subnets to communicate with a single echo server.

What network ports need to be open to run the FreeRTOS port tests?

The following network connections are required to run the FreeRTOS port tests:

Port	Protocol
443, 8883	MQTT
8443	Greengrass Discovery

If you have questions about porting that are not answered on this page or in the rest of the FreeRTOS Porting Guide, please [contact the FreeRTOS engineering team](#).

Downloading FreeRTOS for Porting

Before you begin porting FreeRTOS to your platform, you need to download FreeRTOS or clone the FreeRTOS repository from [GitHub](#). See the [README.md](#) file for instructions.

Note

We recommend that you clone the repository. Cloning makes it easier for you to pick up updates to the master branch as they are pushed to the repository.

After you download or clone FreeRTOS, you can start porting FreeRTOS code to your platform. For instructions, see [Setting Up Your FreeRTOS Source Code for Porting \(p. 9\)](#), and then see [Porting the FreeRTOS Libraries \(p. 25\)](#).

Note

Throughout FreeRTOS documentation, the FreeRTOS download is referred to as `<freertos>`.

Setting Up Your FreeRTOS Source Code for Porting

After you download FreeRTOS, you need to configure some of the files and folders in the FreeRTOS download before you can begin porting.

To prepare your FreeRTOS download for porting, you need to follow the instructions in [Configuring the FreeRTOS Download \(p. 9\)](#) to configure the directory structure of your FreeRTOS download to fit your device.

If you plan to test the ported libraries as you implement them for debugging purposes, you also need to configure some files for testing before you begin porting. For instructions on test set up, see [Setting Up Your FreeRTOS Source Code for Testing \(p. 10\)](#).

Note

You must use the AWS IoT Device Tester for FreeRTOS to officially validate your ports for qualification. For more information about AWS IoT Device Tester for FreeRTOS, see [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide.

For information about qualifying your device for FreeRTOS, see the [FreeRTOS Qualification Guide](#).

After you configure your FreeRTOS download and set up your testing environment, you can begin porting FreeRTOS. For porting and testing instructions, see [Porting the FreeRTOS Libraries \(p. 25\)](#).

Configuring the FreeRTOS Download

Follow the instructions below to configure the FreeRTOS download for porting FreeRTOS code to your device.

Configuring Directories for Vendor-supplied, Board-specific Libraries

Under the download's root directory (`<freertos>`), the vendors folder is structured as follows:

```
vendors
+ - <vendor>      (Template, to be renamed to the name of the MCU vendor)
  + - boards
    |   + - <board>      (Template, to be renamed to the name of the development board)
    |     + - aws_demos
    |     + - aws_tests
    |     + - CMakeLists.txt
    |     + - ports
  + - <driver_library> (Template, to be renamed to the library name)
    + - <driver_library_version> (Template, to be renamed to the library version)
```

The `<vendor>` and `<board>` folders are template folders that we provide to make it easier to create demo and test projects. Their directory structure ensures that all demo and test projects have a consistent organization.

The `aws_tests` folder has the following structure:

```
vendors/<vendor>/boards/<board>/aws_tests
+ - application_code    (Contains main.c, which contains main())
```

```
| + - <vendor>_code    (Contains vendor-supplied, board-specific files)
| + - main.c          (Contains main())
+ - config_files      (Contains FreeRTOS config files)
```

All test projects require vendor-supplied driver libraries. Some vendor-supplied files, such as a header file that maps GPIO output to an LED light, are specific to a target development board. These files belong in the `<vendor>_code` folder.

Other vendor-supplied files, such as a GPIO library, are common across a board's MCU family. These files belong in the `<driver_library>` folder.

To set up the directories for vendor-supplied libraries that are common across an MCU family

1. Save all required vendor-supplied libraries that are common across a target board's MCU family in the `<driver_library_version>` folder.
2. Rename the `<vendor>` folder to the name of the vendor, and rename the `<driver_library>` and `<driver_library_version>` folders to the name of the driver library and its version.

Important

Do not save vendor-supplied libraries that are common across a target board's MCU family to any subdirectories of `<freertos>/test` or `<freertos>/demos`.

Configuring Directories for Project Files

Under `<freertos>`, the `projects` folder is structured as follows:

```
projects
+ - <vendor>          (Template, to be renamed to the name of the MCU vendor)
  + - <board>         (Template, to be renamed to the name of the development board)
    + - <ide>         (Contains an IDE-specific project)
    + - visual_studio (contains project files for Visual Studio)
```

To set up the project directories

1. Rename the `<ide>` folder to the name of the IDE that you are using to build the test project.
2. Rename the `<vendor>` folder to the name of the vendor, and rename the `<board>` folder to the name of the development board.

Configuring FreeRTOSConfig.h

After you have configured the directory structure of your FreeRTOS download, configure your board name in the `FreeRTOSConfig.h` configuration header file.

To configure your board name in FreeRTOSConfig.h

1. Open `<freertos>/venders/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSConfig.h`.
2. In the line `#define configPLATFORM_NAME "<Unknown>"`, change `<Unknown>` to match the name of your board.

Setting Up Your FreeRTOS Source Code for Testing

FreeRTOS includes tests for each ported library. The `aws_test_runner.c` file defines a `RunTests` function that runs each test that you have specified in the `aws_test_runner_config.h` header file.

As you port each FreeRTOS library, you can test the ports by building the ported FreeRTOS source code, flashing the compiled code to your board, and running it on the board.

To build the FreeRTOS source code for testing, you can use either of the following:

- A supported IDE.

If you are using an IDE to build FreeRTOS source code, you need to set up an IDE test project. Follow the instructions in [Creating an IDE Project \(p. 11\)](#) to create a test project in your IDE. Each library-specific porting section under [Porting the FreeRTOS Libraries \(p. 25\)](#) includes additional instructions for setting up a library's ported source files in the IDE test project.

- The [CMake build system](#).

If you are using CMake, you need to create a `CMakeLists.txt` CMake list file. Follow the instructions in [Creating a CMake List File \(p. 12\)](#) to create a `CMakeLists.txt` CMake list file.

Important

A `CMakeLists.txt` file is required for listing a qualified device on the FreeRTOS console, regardless of the testing method that you use.

After you build the code, use your platform's flash utility to flash the compiled code to your device.

Note

You specify your build and flash tools in the `userdata.json` file for Device Tester, so if you are validating your ports with Device Tester, you do not need to flash your code manually.

Creating an IDE Project

After you configure your FreeRTOS download, you can create an IDE project and import the code into the project.

Follow the instructions below to create an IDE project with the required IDE project structure for testing.

Important

If you are using an Eclipse-based IDE, do not configure the project to build all the files in any given folder. Instead, add source files to a project by linking to each source file individually.

1. Open your IDE, and create a project named `aws_tests` in the `freertos/projects/vendor/board/ide` directory.
2. In the IDE, create two virtual folders under the `aws_tests` project:
 - `application_code`
 - `config_files`

Under `aws_tests`, there should now be two virtual folders in the IDE project: `application_code` and `config_files`.

Note

Eclipse generates an additional `includes` folder. This folder is not a part of the required structure.

3. Import all of the files under `freertos/vendors/vendor/boards/board/aws_tests/application_code` and its subdirectories into the `aws_tests/application_code` virtual folder in your IDE.
4. Import all of the files under `freertos/tests` and its subdirectories into the `aws_tests/application_code` virtual directory in your IDE.
5. Import all of the header files in the `freertos/vendors/vendor/boards/board/aws_tests/config_files` directory into the `aws_tests/config_files` virtual folder in your IDE.

Note

If you are not porting a specific library, you do not need to import the files for that library into your project. For example, if you are not porting the OTA library, you can leave out the `aws_ota_agent_config.h` and `aws_test_ota_config.h` files. If you are not porting the Wi-Fi library, you can leave out the `aws_test_wifi_config.h` and `aws_wifi_config.h` files.

6. Import all of the files in `freertos/libraries` and its subdirectories into the `aws_tests` IDE project.

Note

If you are not porting a specific library, you do not need to import the files for that library into your project.

7. Import all of the source files in the `freertos/freertos_kernel` and `freertos/freertos_kernel/include` directories into the `aws_tests` IDE project.
8. Import the subdirectory of `freertos/freertos_kernel/portable` that corresponds to your compiler and platform architecture into the `aws_tests` IDE project.
9. Import the FreeRTOS memory management implementation that you are using for your device into the `aws_tests` IDE project.

The `freertos/freertos_kernel/portable/MemMang` directory contains FreeRTOS memory management implementations. We highly recommend that you use `heap_4.c` or `heap_5.c`.

For more information about FreeRTOS memory management, see [Memory Management](#).

10. Open your project's IDE properties, and add the following paths to your compiler's include path:
 - `freertos/vendors/vendor/boards/board/aws_tests/config_files`
 - `freertos/freertos_kernel/include`
 - `freertos/freertos_kernel/portable/<compiler>/<architecture>`
 - Any paths required for vendor-supplied driver libraries
11. Define `UNITY_INCLUDE_CONFIG_H` and `AMAZON_FREERTOS_ENABLE_UNIT_TESTS` as project-level macros in the project properties.

After you finish setting up your IDE project, you are ready to port the FreeRTOS libraries to your device. For instructions, see [Porting the FreeRTOS Libraries \(p. 25\)](#).

Creating a CMake List File

After you configure your FreeRTOS download for porting, you can set up a CMake list file for your project and platform.

Topics

- [Prerequisites \(p. 12\)](#)
- [Creating a List File for Your Platform from the CMakeLists.txt Template \(p. 13\)](#)
- [Building FreeRTOS with CMake \(p. 20\)](#)

Prerequisites

Make sure that your host machine meets the following prerequisites before you continue:

- Your device's compilation toolchain must support the machine's operating system. CMake supports all versions of Windows, macOS, and Linux.

Windows subsystem for Linux (WSL) is not supported. Use native CMake on Windows machines.

- You must have CMake version 3.13 or later installed.

You can download the binary distribution of CMake from [CMake.org](https://cmake.org).

Note

If you download the binary distribution of CMake, make sure that you add the CMake executable to the PATH environment variable before you use CMake from command line.

You can also download and install CMake using a package manager, like [homebrew](https://brew.sh) on macOS, and [scoop](https://scoop.sh) or [chocolatey](https://chocolatey.org) on Windows.

Note

The CMake package versions in the package managers for many Linux distributions are out-of-date. If your distribution's package manager does not include the the latest version of CMake, you can try [linuxbrew](https://linuxbrew.sh) or [nix](https://nixos.org).

- You must have a compatible native build system.

CMake can target many native build systems, including [GNU Make](https://www.gnu.org/software/make/) or [Ninja](https://ninja-build.org). Both Make and Ninja can be installed with package managers on Linux, macOS, and Windows. If you are using Make on Windows, you can install a standalone version from [Equation](https://equation.co), or you can install [MinGW](https://www.mingw-w64.org), which bundles Make.

Note

The Make executable in MinGW is called `mingw32-make.exe`, instead of `make.exe`.

We recommend that you use Ninja, because it is faster than Make and also provides native support to all desktop operating systems.

Creating a List File for Your Platform from the CMakeLists.txt Template

A `CMakeLists.txt` template file is provided with FreeRTOS, under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt`.

The `CMakeLists.txt` template file consists of four sections:

- [FreeRTOS Console Metadata \(p. 14\)](#)
- [Compiler Settings \(p. 14\)](#)
- [FreeRTOS Portable Layers \(p. 16\)](#)
- [FreeRTOS Demos and Tests \(p. 19\)](#)

Follow the instructions to edit these four sections of the list file to match your platform. You can refer to the `CMakeLists.txt` files for other qualified vendor boards under `<freertos>/vendors` as examples.

Two primary functions are called throughout the file:

```
afr_set_board_metadata(<name> <value>)
```

This function defines metadata for the FreeRTOS console. The function is defined in `<freertos>/tools/cmake/afr_metadata.cmake`.

```
afr_mcu_port(<module_name> [<DEPENDS> [<targets...>]])
```

This function defines the portable-layer target associated with a FreeRTOS module (that is, library). It creates a CMake `GLOBAL INTERFACE IMPORTED` target with a name of the

form `AFR:<module_name>::mcu_port`. If `DEPENDS` is used, additional targets are linked with `target_link_libraries`. The function is defined in `<freertos>/tools/cmake/afr_module.cmake`.

FreeRTOS Console Metadata

The first section of the template file defines the metadata that is used to display a board's information in the FreeRTOS console. Use the function `afr_set_board_metadata(<name> <value>)` to define each field listed in the template. This table provides descriptions of each field.

Field Name	Value Description
ID	A unique ID for the board.
DISPLAY_NAME	The name of the board as you want it displayed on the FreeRTOS console.
DESCRIPTION	A short description of the board for the FreeRTOS console.
VENDOR_NAME	The name of the vendor of the board.
FAMILY_NAME	The name of the board's MCU family.
DATA_RAM_MEMORY	The size of the board's RAM, followed by abbreviated units. For example, use KB for kilobytes.
PROGRAM_MEMORY	The size of the board's program memory, followed by abbreviated units. For example, use "MB" for megabytes.
CODE_SIGNER	The code-signing platform used for OTA updates. Use AmazonFreeRTOS-Default for SHA256 hash algorithm and ECDSA encryption algorithm. If you want to use a different code-signing platform, contact us .
SUPPORTED_IDE	A semicolon-delimited list of IDs for the IDEs that the board supports.
IDE_<ID>_NAME	The name of the supported IDE. Replace <ID> with the ID listed for the IDE in the SUPPORTED_IDE field.
IDE_<ID>_COMPILER	A semicolon-delimited list of names of supported compilers for the supported IDE. Replace <ID> with the ID listed for the IDE in the SUPPORTED_IDE field.

Compiler Settings

The second section of the template file defines the compiler settings for your board. To create a target that holds the compiler settings, call the `afr_mcu_port` function with `compiler` in place of the `<module_name>` to create an `INTERFACE` target with the name `AFR::compiler::mcu_port`. The kernel publicly links to this `INTERFACE` target so that the compiler settings are transitively populated to all modules.

Use the standard, built-in CMake functions to define the compiler settings in this section of the list file. As you define the compiler settings, follow these best practices:

- Use `target_compile_definitions` to provide compile definitions and macros.
- Use `target_compile_options` to provide compiler flags.
- Use `target_include_directories` to provide include directories.
- Use `target_link_options` to provide linker flags.
- Use `target_link_directories` to provide linker-search directories.
- Use `target_link_libraries` to provide libraries to link against.

Note

If you define the compiler settings somewhere else, you don't need to duplicate the information in this section of the file. Instead, call `afr_mcu_port` with `DEPENDS` to bring in the target definition from another location.

For example:

```
# <your_target> is defined somewhere else. It does not have to be in the same file.
afr_mcu_port(compiler DEPENDS <your_target>)
```

When you call `afr_mcu_port` with `DEPENDS`, it calls `target_link_libraries(AFR::<module_name>::mcu_port INTERFACE <your_targets>)`, which populates the compiler settings for the required `AFR::compiler::mcu_port` target.

Using Multiple Compilers

If your board supports multiple compilers, you can use the `AFR_TOOLCHAIN` variable to dynamically select the compiler settings. This variable is set to the name of the compiler you are using, which should be same as the name of the toolchain file found under `<freertos>/tools/cmake/toolchains`.

For example:

```
if("${AFR_TOOLCHAIN}" STREQUAL "arm-gcc")
    afr_mcu_port(compiler DEPENDS my_gcc_settings).
elseif("${AFR_TOOLCHAIN}" STREQUAL "arm-iar")
    afr_mcu_port(compiler DEPENDS my_iar_settings).
else()
    message(FATAL_ERROR "Compiler ${AFR_TOOLCHAIN} not supported.")
endif()
```

Advanced Compiler Settings

If you want to set more advanced compiler settings, such as setting compiler flags based on programming language, or changing settings for different release and debug configurations, you can use CMake generator expressions.

For example:

```
set(common_flags "-foo")
set(c_flags "-foo-c")
set(asm_flags "-foo-asm")
target_compile_options(
    my_compiler_settings INTERFACE
    $<${COMPILE_LANGUAGE:C}:${common_flags} ${c_flags}> # This only have effect on C files.
    $<${COMPILE_LANGUAGE:ASM}:${common_flags} ${asm_flags}> # This only have effect on ASM
    files.
```

```
)
```

CMake generator expressions are not evaluated at the configuration stage, when CMake reads list files. They are evaluated at the generation stage, when CMake finishes reading list files and generates build files for the target build system.

FreeRTOS Portable Layers

The third section of the template file defines all of the portable layer targets for FreeRTOS (that is, libraries).

You must use the `afr_mcu_port(<module_name>)` function to define a portable layer target for each FreeRTOS module that you plan to implement.

You can use any CMake functions you want, as long as the `afr_mcu_port` call creates a target with a name that provides the information required to build the corresponding FreeRTOS module.

The `afr_mcu_port` function creates a **GLOBAL INTERFACE IMPORTED library target** with a name of the form `AFR::<module_name>::mcu_port`. As a **GLOBAL** target, it can be referenced in CMake list files. As an **INTERFACE** target, it is not built as a standalone target or library, but compiled into the corresponding FreeRTOS module. As an **IMPORTED** target, its name includes a namespace (`::`) in the target name (for example, `AFR::kernel::mcu_port`).

Modules without corresponding portable layer targets are disabled by default. If you run CMake to configure FreeRTOS, without defining any portable layer targets, you should see the following output:

```
FreeRTOS modules:
  Modules to build:
  Disabled by user:
  Disabled by dependency:  kernel, posix, pkcs11, secure_sockets, mqtt, ...

  Available demos:
  Available tests:
```

As you update the `CMakeLists.txt` file with porting layer targets, the corresponding FreeRTOS modules are enabled. You should also be able to build any FreeRTOS module whose dependency requirements are subsequently satisfied. For example, if the MQTT library is enabled, the Device Shadow library is also enabled, because its only dependency is the MQTT library.

Note

The FreeRTOS kernel dependency is a minimum requirement. The CMake configuration fails if the FreeRTOS kernel dependency is not satisfied.

Setting Up the Kernel Porting Target

To create the kernel porting target (`AFR::kernel::mcu_port`), call `afr_mcu_port` with the module name `kernel`. When you call `afr_mcu_port`, specify the targets for the FreeRTOS portable layer and driver code. After you create the target, you can provide the dependency information and the FreeRTOS portable layer and driver code information for the target to use.

Follow the these instructions to set up the kernel porting target.

To set up the kernel porting target

1. Create a target for the driver code.

For example, you can create a **STATIC** library target for the driver code:

```
add_library(my_board_driver STATIC ${driver_sources})
```

```
# Use your compiler settings
target_link_libraries(
    my_board_driver
    PRIVATE AFR::compiler::mcu_port
# Or use your own target if you already have it.
# PRIVATE ${compiler_settings_target}
)

target_include_directories(
    my_board_driver
    PRIVATE "<include_dirs_for_private_usage>"
    PUBLIC "<include_dirs_for_public_interface>"
)
```

Or you can create an INTERFACE library target for the driver code:

```
# No need to specify compiler settings since kernel target has them.
add_library(my_board_driver INTERFACE ${driver_sources})
```

Note

An INTERFACE library target does not have build output. If you use an INTERFACE library target, the driver code is compiled into the kernel library.

2. Configure the FreeRTOS portable layer:

```
add_library(freertos_port INTERFACE)
target_sources(
    freertos_port
    INTERFACE
        "${AFR_MODULES_DIR}/freertos_kernel/portable/GCC/ARM_CM4F/port.c"
        "${AFR_MODULES_DIR}/freertos_kernel/portable/GCC/ARM_CM4F/portmacro.h"
        "${AFR_MODULES_DIR}/freertos_kernel/portable/MemMang/heap_4.c"
)
target_include_directories(
    freertos_port
    INTERFACE
        "${AFR_MODULES_DIR}/freertos_kernel/portable/GCC/ARM_CM4F"
        "${include_path_to_FreeRTOSConfig_h}"
)
```

Note

You can also configure the FreeRTOS portable layer by specifying these source files and their include directories directly in the `AFR::kernel::mcu_port` target.

3. Create the kernel portable layer target:

```
# Bring in driver code and freertos portable layer dependency.
afr_mcu_port(kernel DEPENDS my_board_driver freertos_port)

# If you need to specify additional configurations, use standard CMake functions with
# AFR::kernel::mcu_port as the target name.
target_include_directories(
    AFR::kernel::mcu_port
    INTERFACE
        "${additional_includes}" # e.g. board configuration files
)
target_link_libraries(
    AFR::kernel::mcu_port
    INTERFACE
        "${additional_dependencies}"
)
```

4. To test your list file and configuration, you can write a simple application that uses the FreeRTOS kernel port. For more information about developing and building FreeRTOS applications with CMake, see [Building FreeRTOS with CMake \(p. 20\)](#).
5. After you create the demo, add `add_executable` and `target_link_libraries` calls to the list file, and compile the kernel as a static library to verify that the kernel portable layer is correctly configured.

```
add_executable(  
    <my_demo>  
    main.c  
)  
target_link_libraries(  
    <my_demo>  
    PRIVATE AFR::kernel  
)
```

Setting Up the Porting Targets for FreeRTOS Modules

After you add the portable layer target for the kernel, you can add portable layer targets for other FreeRTOS modules.

For example, to add the portable layer for the Wi-Fi module:

```
afr_mcu_port(wifi)  
target_sources(  
    AFR::wifi::mcu_port  
    INTERFACE "${AFR_MODULES_DIR}/vendors/<vendor>/boards/<board>/ports/wifi/aws_wifi.c"  
)
```

This example Wi-Fi module portable layer has only one implementation file, which is based on the driver code.

If you want to add the portable layer for the FreeRTOS Secure Sockets module, the module depends on TLS. This makes its portable layer target slightly more complicated than that of the Wi-Fi module. FreeRTOS provides a default TLS implementation based on mbedTLS that you can link to:

```
afr_mcu_port(secure_sockets)  
target_sources(  
    AFR::secure_sockets::mcu_port  
    INTERFACE ${portable_layer_sources}  
)  
target_link_libraries(  
    AFR::secure_sockets::mcu_port  
    AFR::tls  
)
```

In this example code, the standard CMake function `target_link_libraries` states that the Secure Sockets portable layer depends on `AFR::tls`.

You can reference all FreeRTOS modules by using their target name `AFR::<module_name>`. For example, you can use the same syntax to also state a dependency on FreeRTOS-Plus-TCP:

```
target_link_libraries(  
    AFR::secure_sockets::mcu_port  
    AFR::freertos_plus_tcp  
    AFR::tls  
)
```

Note

If your platform handles TLS by itself, you can use your driver code directly. If you use your driver code directly for TLS, you don't need to call `target_link_libraries`, because all FreeRTOS modules implicitly depend on the kernel that includes your driver code. Because all non-kernel FreeRTOS modules implicitly depend on the kernel, their porting layers don't require you to specify the kernel as a dependency. The POSIX module, however, is defined as an optional kernel module. If you want to use POSIX, you must explicitly include it in your kernel portable layer. For example:

```
# By default, AFR::posix target does not expose standard POSIX headers in its
public
# interface, i.e., You need to use "freertos_plus_posix/source/
FreeRTOS_POSIX_pthread.c" instead of "pthread.h".
# Link to AFR::use_posix instead if you need to use those headers directly.
target_link_libraries(
    AFR::kernel::mcu_port
    INTERFACE AFR::use_posix
)
```

FreeRTOS Demos and Tests

The final section of the template file defines the demo and test targets for FreeRTOS. CMake targets are created automatically for each demo and test that satisfies the dependency requirements.

In this section, define an executable target with the `add_executable` function. Use `aws_tests` as the target name if you're compiling tests, or `aws_demos` if you're compiling demos. You might need to provide other project settings, such as linker scripts and post-build commands. For example:

```
if(AFR_IS_TESTING)
    set(exe_target aws_tests)
else()
    set(exe_target aws_demos)
endif()

set(CMAKE_EXECUTABLE_SUFFIX ".elf")
add_executable(${exe_target} "${board_dir}/application_code/main.c")
```

`target_link_libraries` is then called to link available CMake demo or test targets to your executable target.

Note

You still need to modify `aws_demos/config_files/aws_demo_config.h` and `aws_tests/config_files/aws_test_runner_config.h` to enable demos and tests.

Running Post-build Commands

For information about running post-build commands, see [add_custom_command](#). Use the second signature. For example:

```
# This should run an external command "command --arg1 --arg2".
add_custom_command(
    TARGET ${exe_target} POST_BUILD COMMAND "command" "--arg1" "--arg2"
)
```

Note

CMake supports many common, platform-independent operations for creating directories, copying files, and so on. For more information about CMake command-line operations, see the [CMake command-line tool reference](#). You can reference the CMake command-line tool from a CMake list file with the built-in variable `{CMAKE_COMMAND}`.

Building FreeRTOS with CMake

CMake targets your host operating system as the target system by default. To use CMake for cross compiling, you must provide a toolchain file that specifies the compiler that you want to use. FreeRTOS provides some default toolchain files in `<freertos>/tools/cmake/toolchains`. The instructions for using the toolchain file differ depending on whether you are using the CMake command-line interface or the GUI. [Generating Build Files \(CMake Command-Line Tool\) \(p. 20\)](#) has more details. To learn more about cross-compiling in CMake, visit the [Cross Compiling](#) on the official CMake Wiki.

To build a CMake-based project

1. Run CMake to generate the build files for a native build system, like Make or Ninja.

You can use either the [CMake command-line tool](#) or the [CMake GUI](#) to generate the build files for your native build system.

For information about generating FreeRTOS build files, see [Generating Build Files \(CMake Command-Line Tool\) \(p. 20\)](#) and [Generating Build Files \(CMake GUI\) \(p. 21\)](#).

2. Invoke the native build system to make the project into an executable.

For information about making FreeRTOS build files, see [Building FreeRTOS from Generated Build Files \(p. 23\)](#).

Generating Build Files (CMake Command-Line Tool)

You can use the CMake command-line tool (`cmake`) to generate build files for FreeRTOS from the command line.

To generate the build files, you must specify the target board, compiler and the locations of your source code and build directory. Specify the target board with the `-DVENDOR` option. Specify the compiler with the `-DCOMPILER` option. Specify the location of your source code with the `-S` switch and the location of the generated build files with the `-B` switch.

Note

The compiler must be in the system's `PATH` variable, otherwise you must specify the location of the compiler.

For example, if the vendor is Texas Instruments, and the board is the CC3220 Launchpad, and the compiler is GCC for ARM, you can issue the following command to build from the source files located in the current directory to a directory named `build`:

```
cmake -DVENDOR=ti -DBOARD=cc3220_launchpad -DCOMPILER=arm-ti -S . -B build
```

Note

If you are using Windows, you must specify the native build system because CMake uses Visual Studio by default. For example:

```
cmake -DVENDOR=ti -DBOARD=cc3220_launchpad -DCOMPILER=arm-ti -S . -B build -G Ninja
```

Or:

```
cmake -DVENDOR=ti -DBOARD=cc3220_launchpad -DCOMPILER=arm-ti -S . -B build -G  
"MinGW Makefiles"
```

The regular expressions `${VENDOR}.*` and `${BOARD}.*` are used to search for a matching board, so you don't have to use the full names of the vendor and board for the `VENDOR` and `BOARD` options. Partial

names work, provided there is a single match. For example, the following commands generate the same build files from the same source:

```
cmake -DVENDOR=ti -DCOMPILER=arm-ti -S . -B build
```

```
cmake -DBOARD=cc3220 -DCOMPILER=arm-ti -S . -B build
```

```
cmake -DVENDOR=t -DBOARD=cc -DCOMPILER=arm-ti -S . -B build
```

You can use the `CMAKE_TOOLCHAIN_FILE` option if you want to use a toolchain file that is not located in the default directory `cmake/toolchains`. For example:

```
cmake -DBOARD=cc3220 -DCMAKE_TOOLCHAIN_FILE='/path/to/toolchain_file.cmake' -S . -B build
```

If the toolchain file does not use absolute paths for your compiler, and you didn't add your compiler to the `PATH` environment variable, CMake might not be able to find it. To make sure that CMake finds your toolchain file, you can use the `AFR_TOOLCHAIN_PATH` option. This option searches the specified toolchain directory path and the toolchain's subfolder under `bin`. For example:

```
cmake -DBOARD=cc3220 -DCMAKE_TOOLCHAIN_FILE='/path/to/toolchain_file.cmake' -  
DAFR_TOOLCHAIN_PATH='/path/to/toolchain/' -S . -B build
```

To enable debugging, set the `CMAKE_BUILD_TYPE` to `debug`. With this option enabled, CMake adds debug flags to the compile options, and builds FreeRTOS with debug symbols.

```
# Build with debug symbols  
cmake -DBOARD=cc3220 -DCOMPILER=arm-ti -DCMAKE_BUILD_TYPE=debug -S . -B build
```

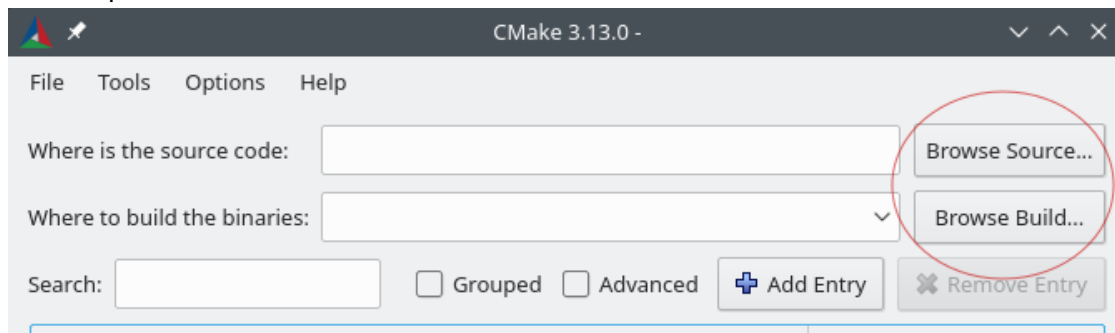
You can also set the `CMAKE_BUILD_TYPE` to `release` to add optimization flags to the compile options.

Generating Build Files (CMake GUI)

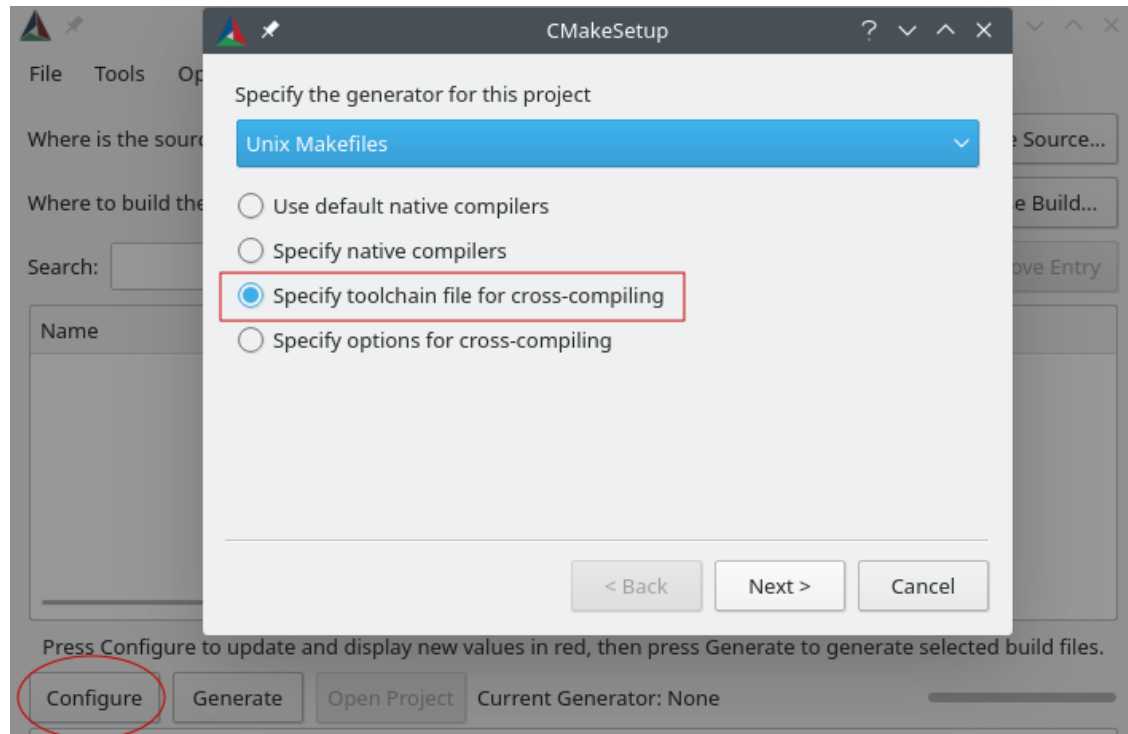
You can use the CMake GUI to generate FreeRTOS build files.

To generate build files with the CMake GUI

1. From the command line, issue `cmake-gui` to start the GUI.
2. Choose **Browse Source** and specify the source input, and then choose **Browse Build** and specify the build output.



3. Choose **Configure**, and under **Specify the build generator for this project**, find and choose the build system that you want to use to build the generated build files.



Note

If you do not see the pop up window, you might be reusing an existing build directory. In this case, delete the CMake cache first by clicking **File->Delete** Cache in the menu.

4. Choose **Specify toolchain file for cross-compiling**, and then choose **Next**.
5. Choose the toolchain file (for example, `<freertos>/tools/cmake/toolchains/arm-ti.cmake`), and then choose **Finish**.

The default configuration for FreeRTOS is the template board, which does not provide any portable layer targets. As a result, a window appears with the message **Error in configuration process**.

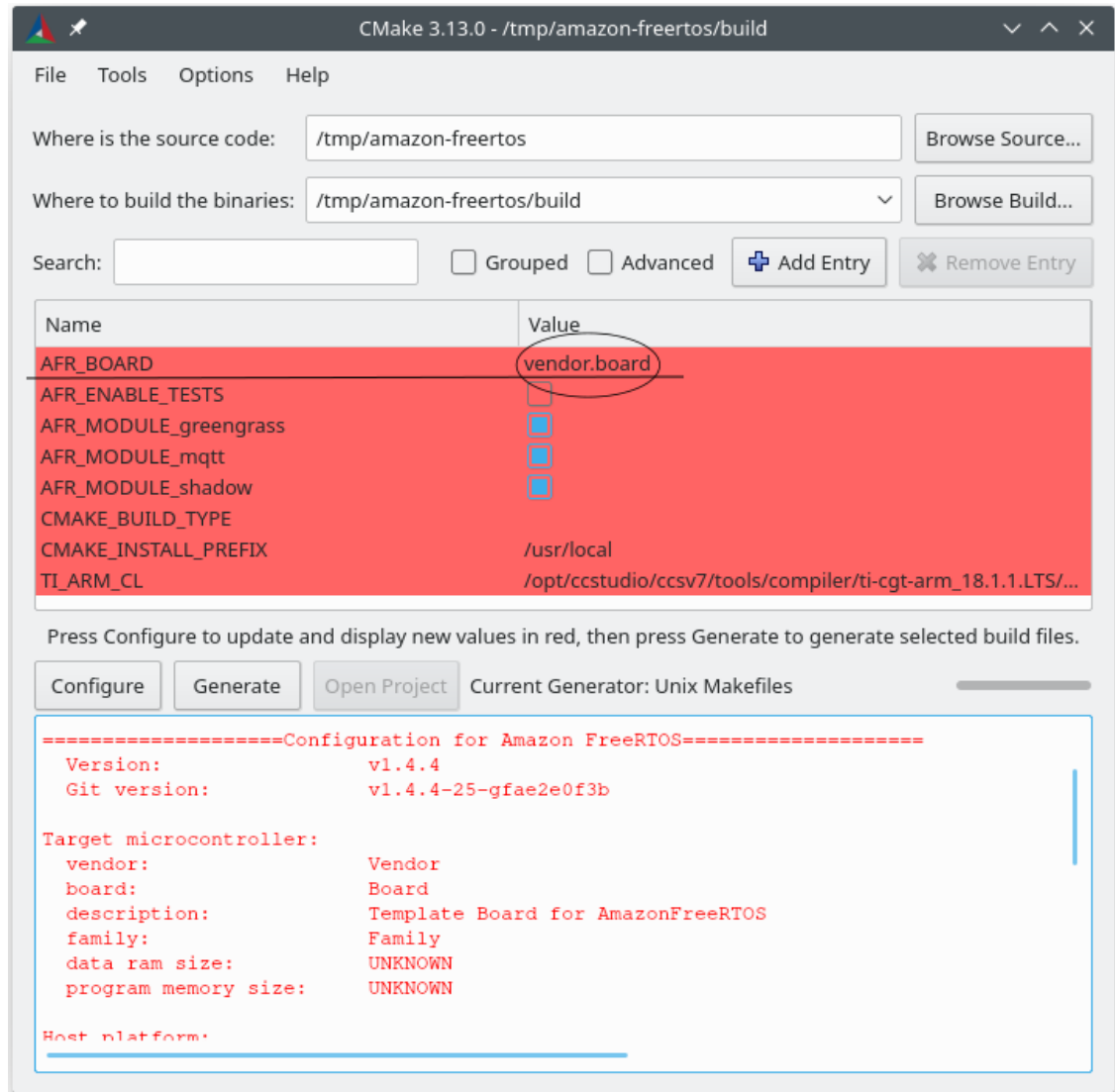
Note

If you see the following error message:

```
CMake Error at tools/cmake/toolchains/find_compiler.cmake:23 (message):
Compiler not found, you can specify search path with "AFR_TOOLCHAIN_PATH".
```

It means the compiler is not in your environment variable PATH. You can set the AFR_TOOLCHAIN_PATH variable in the GUI to tell CMake where you installed your compiler. If you do not see the AFR_TOOLCHAIN_PATH variable, click the **Add Entry** button in the pop up window, enter AFR_TOOLCHAIN_PATH as the **name**, select PATH as the **type**, and enter the compiler path in the **value**, for example, "C:/toolchains/arm-none-eabi-gcc".

6. The GUI should now look like this:



Choose **AFR_BOARD**, choose your board, and then choose **Configure** again.

7. Choose **Generate**. CMake generates the build system files (for example, makefiles or ninja files), and these files appear in the build directory you specified in the first step. Follow the instructions in the next section to generate the binary image.

Building FreeRTOS from Generated Build Files

You can build FreeRTOS with a native build system by calling the build system command from the output binaries directory. For example, if your build file output directory is `build`, and you are using Make as your native build system, run the following commands:

```
cd build
make -j4
```

You can also use the CMake command-line tool to build FreeRTOS. CMake provides an abstraction layer for calling native build systems. For example:

```
cmake --build <build_dir>
```

Here are some other common uses of the CMake command-line tool's build mode:

```
# Take advantage of CPU cores.  
cmake --build <build_dir> --parallel 8
```

```
# Build specific targets.  
cmake --build <build_dir> --target afr_kernel
```

```
# Clean first, then build.  
cmake --build <build_dir> --clean-first
```

For more information about the CMake build mode, see the [CMake documentation](#).

Porting the FreeRTOS Libraries

Before you start porting, follow the instructions in [Setting Up Your FreeRTOS Source Code for Porting](#) (p. 9).

To port FreeRTOS to your device, follow the instructions in the topics below.

1. [Implementing the `configPRINTF_STRING\(\)` macro](#) (p. 27)
2. [Configuring a FreeRTOS Kernel Port](#) (p. 28)
3. [Porting the Wi-Fi Library](#) (p. 29)

Note

If your device does not support Wi-Fi, you can use an ethernet connection to connect to the AWS Cloud instead. A port of the FreeRTOS Wi-Fi library is not necessarily required.

4. [Porting a TCP/IP Stack](#) (p. 35)
5. [Porting the Secure Sockets Library](#) (p. 39)
6. [Porting the PKCS #11 Library](#) (p. 45)
7. [Porting the TLS Library](#) (p. 49)
8. [Configuring the MQTT Library for Testing](#) (p. 67)
9. [Configuring the HTTPS Client Library for Testing](#) (p. 68)

Note

Currently, a port of the FreeRTOS HTTPS library is not required for qualification.

- 10 [Porting the OTA Library](#) (p. 69)

Note

Currently, a port of the FreeRTOS OTA update library is not required for qualification.

- 11 [Porting the Bluetooth Low Energy Library](#) (p. 76)

Note

Currently, a port of the FreeRTOS Bluetooth Low Energy library is not required for qualification.

- 12 [Porting the Common I/O Libraries](#) (p. 80)

Note

Currently, a port of the FreeRTOS common I/O library is not required for qualification.

After you port FreeRTOS to your board, you can officially validate the ports for FreeRTOS qualification with AWS IoT Device Tester for FreeRTOS. For more information about AWS IoT Device Tester for FreeRTOS, see [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide.

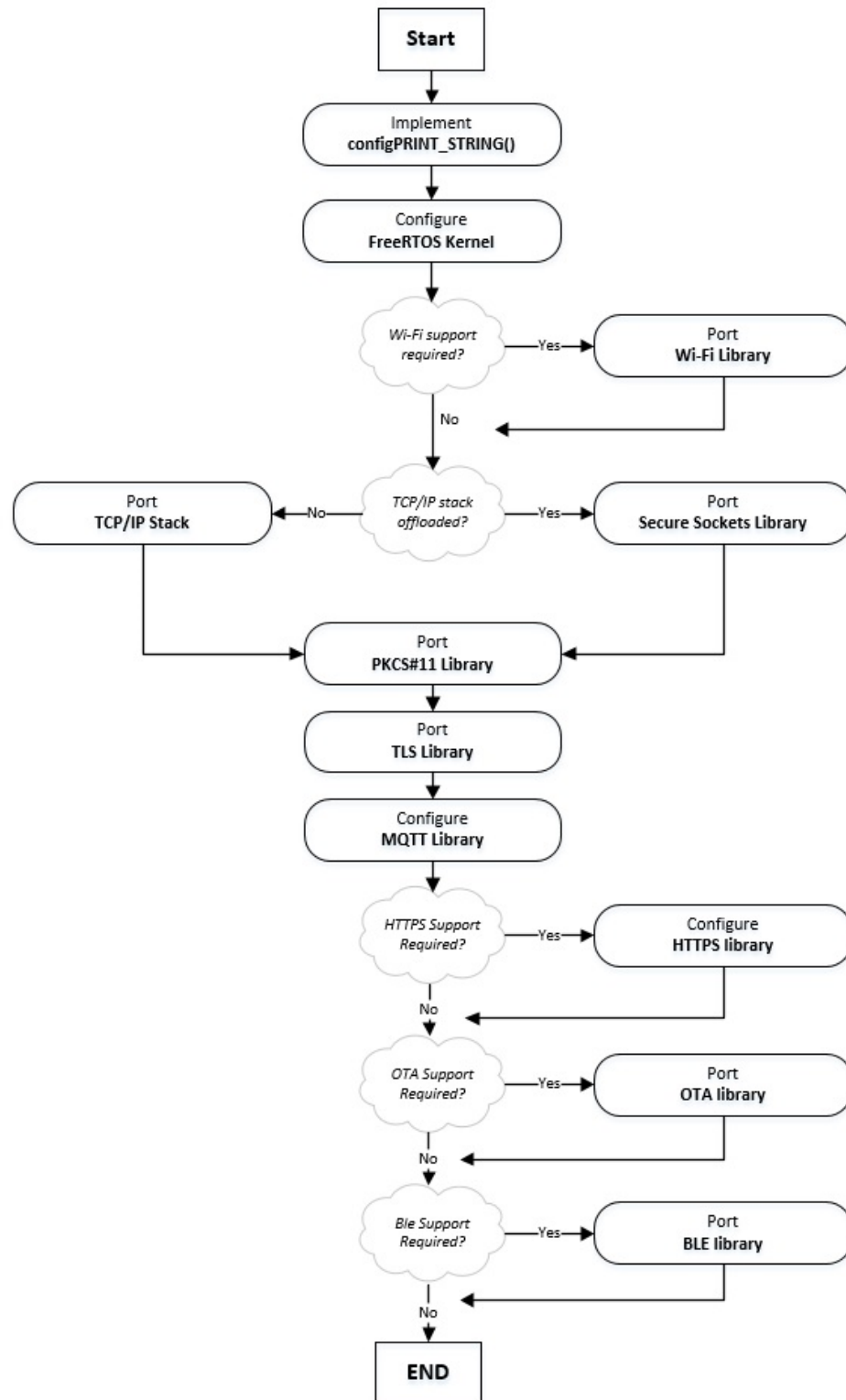
Note

Before you validate your port of FreeRTOS using the AWS IoT Device Tester for FreeRTOS you must remove any logging messages that you might have inserted in your code for testing purposes, otherwise the validation may fail.

For information about qualifying your device for FreeRTOS, see the [FreeRTOS Qualification Guide](#).

FreeRTOS Porting Flowchart

Use the flowchart below for visual aid as you port FreeRTOS to your device.



Implementing the configPRINT_STRING() macro

You must implement the configPRINT_STRING() macro before you port the FreeRTOS libraries. FreeRTOS uses configPRINT_STRING() to output test results as human-readable ASCII strings.

Prerequisites

To implement the configPRINT_STRING() macro, you need the following:

- A development board that supports UART or virtual COM port output.
- A FreeRTOS project configured for your platform, and a porting-test IDE project.

For information, see [Setting Up Your FreeRTOS Source Code for Porting \(p. 9\)](#).

Implementation

To implement configPRINT_STRING()

1. Connect your device to a terminal emulator to output test results.
2. Open the file `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/application_code/main.c`, and locate the call to `configPRINT_STRING("Test Message")` in the `prvMiscInitialization()` function.
3. Immediately before the call to `configPRINT_STRING("Test Message")`, add code that uses the vendor-supplied UART driver to initialize the UART baud rate level to 115200.
4. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSConfig.h`, and locate the empty definition of `configPRINT_STRING()`. The macro takes a NULL-terminated ASCII C string as its only parameter.
5. Update the empty definition of `configPRINT_STRING()` so that it calls the vendor-supplied UART output function.

For example, suppose the UART output function has the following prototype:

```
void MyUARTOutput( char *DataToOutput, size_t LengthToOutput );
```

You would implement `configPRINT_STRING()` as:

```
#define configPRINT_STRING( X ) MyUARTOutput( (X), strlen( (X) ) )
```

Testing

Build and execute the test demo project. If `Test Message` appears in the UART console, then the console is connected and configured correctly, `configPRINT_STRING()` is behaving properly, and testing is complete. You can remove the call to `configPRINT_STRING("Test Message")` from `prvMiscInitialization()`.

After you implement the `configPRINT_STRING()` macro, you can start configuring a FreeRTOS kernel port for your device. See [Configuring a FreeRTOS Kernel Port \(p. 28\)](#) for instructions.

Configuring a FreeRTOS Kernel Port

This section provides instructions for integrating a port of the FreeRTOS kernel into a FreeRTOS port-testing project. For a list of available kernel ports, see [FreeRTOS Kernel Ports](#).

FreeRTOS uses the FreeRTOS kernel for multitasking and inter-task communications. For more information, see the [FreeRTOS Kernel Fundamentals](#) in the FreeRTOS User Guide and [FreeRTOS.org](#).

Note

Porting the FreeRTOS kernel to a new architecture is out of the scope of this documentation. If you are interested in porting the FreeRTOS kernel to a new architecture, [contact the FreeRTOS engineering team](#).

For the FreeRTOS Qualification program, only existing ports are supported. Modifications to these ports are not accepted within the Qualification program. Only the official ports that can be downloaded from [Github](#) or [Sourceforge](#) are accepted.

Prerequisites

To set up the FreeRTOS kernel for porting, you need the following:

- An official FreeRTOS kernel port for the target platform.
- An IDE project or CMakeLists.txt list file that includes the correct FreeRTOS kernel port files for the target platform and compiler.

For information about setting up a test project, see [Setting Up Your FreeRTOS Source Code for Porting](#) (p. 9).

- An implementation of the `configPRINTF_STRING()` macro for your device.

For information about implementing `configPRINTF_STRING()`, see [Implementing the configPRINTF_STRING\(\) macro](#) (p. 27).

Configuring the FreeRTOS Kernel

The header file `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSConfig.h` specifies application-specific configuration settings for the FreeRTOS kernel. For a description of each configuration option, see [Customisation](#) on FreeRTOS.org.

To configure the FreeRTOS kernel to work with your device, open `FreeRTOSConfig.h`, and verify that the configuration options in the following table are correctly specified for your platform.

Configuration option	Description
<code>configCPU_CLOCK_HZ</code>	Specifies the frequency of the clock used to generate the tick interrupt.
<code>configMINIMAL_STACK_SIZE</code>	Specifies the minimum stack size. As a starting point, this can be set to the value used in the official FreeRTOS demo for the FreeRTOS kernel port in use. Official FreeRTOS demos are those distributed from the FreeRTOS.org website. Make sure that stack overflow checking is set to 2, and increase <code>configMINIMAL_STACK_SIZE</code> if overflows occur. To save RAM, set stack sizes to the minimum value that does not result in a stack overflow.

Configuration option	Description
<code>configTOTAL_HEAP_SIZE</code>	Sets the size of the FreeRTOS heap . Like task stack sizes, the heap size can be tuned to ensure unused heap space does not consume RAM.

Note

If you are porting ARM Cortex-M3, M4, or M7 devices, you must also specify `configPRIO_BITS` and `configMAX_SYSCALL_INTERRUPT_PRIORITY` correctly.

Testing

1. Open `/libraries/freertos_plus/standard/utils/src/iot_system_init.c`, and comment out the lines that call `BUFFERPOOL_Init()`, `MQTT_AGENT_Init()`, and `SOCKETS_Init()` from within function `SYSTEM_Init()`. These initialization functions belong to libraries that you haven't ported yet. The porting sections for those libraries include instructions to uncomment these functions.
2. Build the test project, and then flash it to your device for execution.
3. If "." appears in the UART console every 5 seconds, then the FreeRTOS kernel is configured correctly, and testing is complete.

Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSConfig.h`, and set `configUSE_IDLE_HOOK` to 0 to stop the kernel from executing `vApplicationIdleHook()` and outputting ".".

4. If "." appears at any frequency other than 5 seconds, open `FreeRTOSConfig.h` and verify that `configCPU_CLOCK_HZ` is set to the correct value for your board.

After you have configured the FreeRTOS kernel port for your device, you can start porting the Wi-Fi library. See [Porting the Wi-Fi Library \(p. 29\)](#) for instructions.

Porting the Wi-Fi Library

The FreeRTOS Wi-Fi library interfaces with vendor-supplied Wi-Fi drivers. For more information about the FreeRTOS Wi-Fi library, see [FreeRTOS Wi-Fi Library](#) in the FreeRTOS User Guide.

If your device does not support Wi-Fi networking, you can skip porting the FreeRTOS Wi-Fi library and start [Porting a TCP/IP Stack \(p. 35\)](#).

Note

For qualification, your device must connect to the AWS Cloud. If your device does not support Wi-Fi, you can use an ethernet connection instead. A port of the FreeRTOS Wi-Fi library is not necessarily required.

Prerequisites

To port the Wi-Fi library, you need the following:

- An IDE project or `CMakeLists.txt` list file that includes the vendor-supplied Wi-Fi drivers.

For information about setting up a test project, see [Setting Up Your FreeRTOS Source Code for Porting \(p. 9\)](#).

- A validated configuration of the FreeRTOS kernel.

For information about configuring the FreeRTOS kernel for your platform, see [Configuring a FreeRTOS Kernel Port \(p. 28\)](#).

- Two wireless access points.

Porting

`<freertos>/vendors/<vendor>/boards/<board>/ports/wifi/aws_wifi.c` contains empty definitions of a set of Wi-Fi management functions. Use the vendor-supplied Wi-Fi driver library to implement at least the set of functions listed in the following table.

Function	Description
WIFI_On	Turns on Wi-Fi module and initializes the drivers.
WIFI_ConnectAP	Connects to a Wi-Fi access point (AP).
WIFI_Disconnect	Disconnects from an AP.
WIFI_Scan	Performs a Wi-Fi network scan.
WIFI_GetIP	Retrieves the Wi-Fi interface's IP address.
WIFI_GetMAC	Retrieves the Wi-Fi interface's MAC address.
WIFI_GetHostIP	Retrieves the host IP address from a hostname using DNS.

`<freertos>/libraries/abstractions/wifi/include/aws_wifi.h` provides the information required to implement these functions.

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the Wi-Fi library in the IDE project

1. Add the source file `<freertos>/vendors/<vendor>/boards/<board>/ports/wifi/aws_wifi.c` to your `aws_tests` IDE project.
2. Add the source file `aws_test_wifi.c` to the `aws_tests` IDE project.

Configuring the `CMakeLists.txt` File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

See below for an example portable layer target definition for the Wi-Fi library.

```
# WiFi
afr_mcu_port(wifi)
target_sources(
    AFR:wifi:mcu_port
    INTERFACE "<freertos>/vendors/<vendor>/boards/<board>/ports/wifi/aws_wifi.c"
)
```

Setting Up Your Local Testing Environment

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the Wi-Fi tests

1. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/application_code/main.c`, and delete the `#if 0` and `#endif` compiler directives in the function definitions of `vApplicationDaemonTaskStartupHook(void)` and `prvWifiConnect(void)`.
2. Open `<freertos>/libraries/freertos_plus/standard/utils/src/iot_system_init.c`, and in the function `SYSTEM_Init()`, comment out the lines that call `BUFFERPOOL_Init()` and `MQTT_AGENT_Init()`, if you have not done so already. Bufferpool and the MQTT agent are not used in this library's porting tests. When you reach the [Configuring the MQTT Library for Testing \(p. 67\)](#) section, you will be instructed to uncomment these initialization function calls for testing the MQTT library.

If you have not ported the Secure Sockets library, also comment out the line that calls `SOCKETS_Init()`. When you reach the [Porting the Secure Sockets Library \(p. 39\)](#) section, you will be instructed to uncomment this initialization function call.

3. Open `<freertos>/tests/include/aws_clientcredential.h`, and set the macros in the following table for the first AP.

Macro	Value
<code>clientcredentialWIFI_SSID</code>	The Wi-Fi SSID as a C string (in quotation marks).
<code>clientcredentialWIFI_PASSWORD</code>	The Wi-Fi password as a C string (in quotation marks).
<code>clientcredentialWIFI_SECURITY</code>	One of the following: <ul style="list-style-type: none">• <code>eWiFiSecurityOpen</code>• <code>eWiFiSecurityWEP</code>• <code>eWiFiSecurityWPA</code>• <code>eWiFiSecurityWPA2</code> <code>eWiFiSecurityWPA2</code> is recommended.

4. Open `<freertos>/libraries/abstractions/wifi/test/aws_test_wifi.h`, and set the macros in the following table for the second AP.

Macro	Value
testWIFI_SSID	The Wi-Fi SSID as a C string (in quotation marks).
testWIFI_PASSWORD	The Wi-Fi password as a C string (in quotation marks).
testWIFI_SECURITY	One of the following: <ul style="list-style-type: none">• eWiFiSecurityOpen• eWiFiSecurityWEP• eWiFiSecurityWPA• eWiFiSecurityWPA2 eWiFiSecurityWPA2 is recommended.

5. To enable the Wi-Fi tests, open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_WIFI_ENABLED` to 1.

Important

The following tests require a port of the Secure Sockets library and a running echo server:

- WiFiConnectionLoop
- WiFiIsConnected
- WiFiConnectMultipleAP

You won't be able to pass these tests until you port the Secure Sockets library and start an echo server. After you port the Secure Sockets library and start an echo server, rerun the Wi-Fi tests to be sure that all tests pass. For information about porting the Secure Sockets library, see [Porting the Secure Sockets Library \(p. 39\)](#). For information about setting up an echo server, see [Setting Up an Echo Server \(p. 43\)](#).

Running the Tests

To execute the Wi-Fi tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console.

```
.....-----STARTING TESTS-----
UUID: _uuid_d46442f6f32f4759bbfbefc66296069e
.4 29881 [TestRunner] WiFi Networks and strength:
5 29885 [TestRunner]   MikroTik-11AA97: -51
6 29889 [TestRunner]   afrlab-test-ubi_IoT: -41
7 29894 [TestRunner]   afrlab-test-ubi-mtk_IoT: -55
8 29899 [TestRunner]   afrlab-guest-mtk: -55
9 29903 [TestRunner]   afrlab-guest: -41
10 29907 [TestRunner]   afrlab-test: -41
11 29911 [TestRunner]   ConfigureMe: -52
12 29915 [TestRunner]   afrlab-test-mtk: -57
13 29919 [TestRunner]   : -39
14 29922 [TestRunner]   wpa2: -78
15 29926 [TestRunner] End of WiFi Networks
.TEST(Full_WiFi, AFQP_WiFiOnOff) PASS
..16 34778 [TestRunner] WiFi Networks and strength:
17 34782 [TestRunner]   ConfigureMe: -51
18 34786 [TestRunner]   afrlab-test-ubi-mtk_IoT: -52
19 34791 [TestRunner]   afrlab-guest-mtk: -52
20 34796 [TestRunner]   afrlab-test-ubi_IoT: -40
21 34800 [TestRunner]   afrlab-test-mtk: -51
22 34805 [TestRunner]   afrlab-guest: -40
23 34809 [TestRunner]   afrlab-test: -40
24 34813 [TestRunner]   MikroTik-11AA97: -51
25 34817 [TestRunner]   WifiSensorL16: -84
26 34821 [TestRunner]   : -41
27 34824 [TestRunner] End of WiFi Networks
.TEST(Full_WiFi, AFQP_WiFiMode) PASS
```

...

```
513 603362 [TestRunner] End of WiFi Networks
.TEST(Full_WiFi, AFQP_WIFI_NetworkAdd_AddManyNetworks) PASS
.514 608223 [TestRunner] WiFi Networks and strength:
515 608228 [TestRunner]   ConfigureMe: -49
516 608232 [TestRunner]   afrlab-test-ubi-mtk_IoT: -49
517 608237 [TestRunner]   afrlab-test-mtk: -49
518 608241 [TestRunner]   afrlab-guest-mtk: -49
519 608246 [TestRunner]   afrlab-guest: -40
520 608250 [TestRunner]   afrlab-test-ubi_IoT: -40
521 608255 [TestRunner]   afrlab-test: -40
522 608259 [TestRunner]   : -41
523 608262 [TestRunner]   IoTDeviceServicesLab: -46
524 608267 [TestRunner]   IoTDeviceServicesLab100: -45
525 608273 [TestRunner] End of WiFi Networks
.TEST(Full_WiFi, AFQP_WIFI_NetworkDelete_DeleteManyNetworks) PASS
..526 613134 [TestRunner] WiFi Networks and strength:
527 613139 [TestRunner]   ConfigureMe: -49
528 613143 [TestRunner]   afrlab-test-ubi_IoT: -40
529 613148 [TestRunner]   afrlab-test-mtk: -49
530 613152 [TestRunner]   afrlab-guest: -40
531 613156 [TestRunner]   afrlab-test: -40
532 613161 [TestRunner]   WifiSensorL16: -85
533 613165 [TestRunner]   afrlab-test-ubi-mtk_IoT: -51
534 613170 [TestRunner]   afrlab-guest-mtk: -49
535 613175 [TestRunner]   : -39
536 613178 [TestRunner]   wpa2: -68
537 613182 [TestRunner] End of WiFi Networks
.....TEST(Full_WiFi, AFQP_WIFI_ConnectAP_ConnectAllChannels) PASS
....538 666067 [TestRunner] Wi-Fi reconnected following tests finished.

-----
43 Tests 0 Failures 0 Ignored
```

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the device.json file in the Device Tester configs folder.

After you finish porting the FreeRTOS Wi-Fi library to your device, you can start porting a TCP/IP stack. See [Porting a TCP/IP Stack \(p. 35\)](#) for instructions.

Porting a TCP/IP Stack

FreeRTOS provides a TCP/IP stack for boards that do not have on-chip TCP/IP functionality. If your platform offloads TCP/IP functionality to a separate network processor or module, you can skip this porting section and start [Porting the Secure Sockets Library \(p. 39\)](#).

FreeRTOS+TCP is a native TCP/IP stack for the FreeRTOS kernel. FreeRTOS+TCP is maintained by the FreeRTOS engineering team and is the recommended TCP/IP stack to use with FreeRTOS. For more information, see [Porting FreeRTOS+TCP \(p. 35\)](#).

The lightweight IP (lwIP) TCP/IP stack is an open source third-party TCP/IP stack, ported to the FreeRTOS kernel. The lwIP port layer currently supports lwIP version 2.03. For more information, see [Porting lwIP \(p. 38\)](#).

Note

These porting sections only provide instructions for porting to a platform's Ethernet or Wi-Fi driver. The tests only ensure that the Ethernet or Wi-Fi driver can connect to a network. You cannot test sending and receiving data across a network until you have ported the Secure Sockets library.

Porting FreeRTOS+TCP

FreeRTOS+TCP is a native TCP/IP stack for the FreeRTOS kernel. For more information, see [FreeRTOS.org](#).

Prerequisites

To port the FreeRTOS+TCP library, you need the following:

- An IDE project or `CMakeLists.txt` list file that includes the vendor-supplied Ethernet or Wi-Fi drivers.

For information about setting up a test project, see [Setting Up Your FreeRTOS Source Code for Porting \(p. 9\)](#).

- A validated configuration of the FreeRTOS kernel.

For information about configuring the FreeRTOS kernel for your platform, see [Configuring a FreeRTOS Kernel Port \(p. 28\)](#).

Porting

Before you start porting the FreeRTOS-TCP library, check the `<freertos>/libraries/freertos_plus/standard/freertos_plus_tcp/source/portable/NetworkInterface` directory to see if a port to your device already exists.

If a port does not exist, do the following:

1. Follow the [Porting FreeRTOS+TCP to a Different Microcontroller](#) instructions on FreeRTOS.org to port FreeRTOS+TCP to your device.
2. If necessary, follow the [Porting FreeRTOS+TCP to a New Embedded C Compiler](#) instructions on FreeRTOS.org to port FreeRTOS+TCP to a new compiler.

3. Implement a new port that uses the vendor-supplied Ethernet or Wi-Fi drivers in a file called `NetworkInterface.c`, and save the file to `<freertos>/libraries/freertos_plus/standard/freertos_plus_tcp/source/portable/NetworkInterface/<board_family>`.

Note

The files in the `<freertos>/libraries/freertos_plus/standard/freertos_plus_tcp/source/portable/BufferManagement` directory are used by multiple ports. Do not edit the files in this directory.

After you create a port, or if a port already exists, open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSIPConfig.h`, and edit the configuration options so they are correct for your platform. For more information about the configuration options, see [FreeRTOS+TCP Configuration](#) on FreeRTOS.org.

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the FreeRTOS+TCP library in the IDE project

1. Add all of the source and header files in `<freertos>/libraries/freertos_plus/standard/freertos_plus_tcp` and its subdirectories to the `aws_tests` IDE project.

Note

FreeRTOS includes five example heap management implementations under `<freertos>/freertos_kernel/portable/MemMang`. FreeRTOS+TCP and `BufferAllocation_2.c` require the `heap_4.c` or `heap_5.c` implementations. You must use `heap_4.c` or `heap_5.c` to ensure that the FreeRTOS demo applications run properly. Do not use a custom heap implementation.

2. Add `<freertos>/libraries/freertos_plus/standard/freertos_plus_tcp/include` to your compiler's include path.

Configuring the `CMakeLists.txt` File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers](#) (p. 16).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

Setting Up Your Local Testing Environment

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the TCP tests

1. Open `<freertos>/libraries/freertos_plus/standard/utils/src/iot_system_init.c`, and in the function `SYSTEM_Init()`, comment out the lines that call `BUFFERPOOL_Init()` and `MQTT_AGENT_Init()`, if you have not done so already. Bufferpool and the MQTT agent are not used in this library's porting tests. When you reach the [Configuring the MQTT Library for Testing \(p. 67\)](#) section, you will be instructed to uncomment these initialization function calls for testing the MQTT library.

If you have not ported the Secure Sockets library, also comment out the line that calls `SOCKETS_Init()`. When you reach the [Porting the Secure Sockets Library \(p. 39\)](#) section, you will be instructed to uncomment this initialization function call.

2. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/application_code/main.c`, and uncomment the call to `FreeRTOS_IPInit()`.
3. Fill the following arrays with valid values for your network:

Variable	Description
<code>uint8_t ucMACAddress[6]</code>	Default MAC address configuration.
<code>uint8_t ucIPAddress[4]</code>	Default IP address configuration. Note By default, the IP address is acquired by DHCP. If DHCP fails or if you do not want to use DHCP, the static IP address that is defined here is used. To disable DHCP, open <code><freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSIPConfig.h</code> , and set <code>ipconfigUSE_DHCP</code> to 0.
<code>uint8_t ucNetMask[4]</code>	Default net mask configuration.
<code>uint8_t ucGatewayAddress[4]</code>	Default gateway address configuration.
<code>uint8_t ucDNSServerAddress[4]</code>	Default DNS server address configuration.

4. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/FreeRTOSIPConfig.h`, and set the `ipconfigUSE_NETWORK_EVENT_HOOK` macro to 1.
5. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/application_code/main.c`, and add the following code to the beginning of the function definition for `vApplicationIPNetworkEventHook()`:

```
if (eNetworkEvent == eNetworkUp)
{
    configPRINTF("Network connection successful. \n\r");
}
```

Running the Tests

To execute the FreeRTOS+TCP tests

1. Build the test project, and then flash it to your device for execution.

2. Check the test results in the UART console. If `Network connection successful` appears, the Ethernet or Wi-Fi driver successfully connected to the network, and the test is complete.

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the `device.json` file in the Device Tester configs folder.

Porting lwIP

lwIP is an alternative, open source TCP/IP stack. For more information, see [lwIP - A Lightweight TCP/IP Stack - Summary](#). FreeRTOS currently supports version 2.1.2.

Prerequisites

To port the lwIP stack, you need the following:

- An IDE project or `CMakeLists.txt` list file that includes vendor-supplied network drivers.
- A validated configuration of the FreeRTOS kernel.

For information about configuring the FreeRTOS kernel for your platform, see [Configuring a FreeRTOS Kernel Port \(p. 28\)](#).

Porting

Before you port the lwIP TCP/IP stack to your device, check the `<freertos>/libraries/3rdparty/lwip/src/portable` directory to see if a port to your platform already exists.

1. If a port does not exist, do the following:

Under `<freertos>/libraries/3rdparty/lwip/src/portable`, create a directory named `<vendor>/<board>/netif`, where the `<vendor>` and `<board>` directories match your platform.

2. Port the `<freertos>/libraries/3rdparty/lwip/src/netif/ethernetif.c` stub file according to the comments in the stub file.
3. After you have created a port, or if a port already exists, in the test project's `main.c` file, add a call to `tcpip_init()`.
4. In `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files`, create a configuration file named `lwipopts.h`. This file must contain the following line:

```
#include "arch/lwipopts_freertos.h"
```

The file should also contain any platform-specific configuration options.

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Note

There are no TCP/IP porting tests specific to lwIP.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the lwIP source files in the IDE project

1. Add all of the source and header files in `lwip/src` and its subdirectories to the `aws_tests` IDE project.

Note

If you added a `.c` file to the IDE project, and then edited that `.c` file for a port, you must replace the original `.c` file with the edited one in the IDE project.

2. Add the following paths to your compiler's include path:
 - `<freertos>/libraries/3rdparty/lwip/src/include`
 - `<freertos>/libraries/3rdparty/lwip/src/portable`
 - `<freertos>/libraries/3rdparty/lwip/src/portable/<vendor>/<board>/include`

Configuring the CMakeLists.txt File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers](#) (p. 16).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

A Secure Sockets library implementation already exists for the FreeRTOS+TCP TCP/IP stack and the lwIP stack. If you are using FreeRTOS+TCP or lwIP, you do not need to port the Secure Sockets library. After you finish porting the FreeRTOS+TCP stack or the lwIP stack to your device, you can start [Porting the PKCS #11 Library](#) (p. 45). Even if you do not need to create a port for the Secure Sockets library, your platform still needs to pass the AWS IoT Device Tester tests for the Secure Sockets library for qualification.

Porting the Secure Sockets Library

You can use the FreeRTOS Secure Sockets library to create and configure a TCP socket, connect to an MQTT broker, and send and receive TCP data. The Secure Sockets library also encapsulates TLS functionality. Only a standard TCP socket is required to create a TLS-protected socket. For more information, see [FreeRTOS Secure Sockets Library](#) in the FreeRTOS User Guide.

FreeRTOS includes a Secure Sockets implementation for the [FreeRTOS+TCP](#) and [lightweight IP \(lwIP\)](#) TCP/IP stacks, which are used in conjunction with [mbedtls](#). If you are using either the FreeRTOS+TCP or the lwIP TCP/IP stack, you do not need to port the Secure Sockets library.

Note

Even if you do not need to create a port of the Secure Sockets library, your platform must still pass the qualification tests for the Secure Sockets library. Qualification is based on results from AWS IoT Device Tester.

If your platform offloads TCP/IP functionality to a separate network chip, you need to port the FreeRTOS Secure Sockets library to your device.

Prerequisites

To port the Secure Sockets library, you need the following:

- A port of the Wi-Fi library (required only if you are using Wi-Fi for network connectivity).

For information about porting the Wi-Fi library, see [Porting the Wi-Fi Library \(p. 29\)](#).

- A port of a TCP/IP stack.

For information about porting a TCP/IP stack, see [Porting a TCP/IP Stack \(p. 35\)](#).

- An echo server.

FreeRTOS includes an echo server, written in Go, in the `<freertos>/tools/echo_server` directory. For more information, see [Setting Up an Echo Server \(p. 43\)](#).

Porting

If your platform offloads TCP/IP functionality to a separate network chip, you need to implement all the functions for which stubs exist in `<freertos>/vendors/<vendor>/boards/<board>/ports/secure_sockets/iot_secure_sockets.c`.

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the Secure Sockets library in the IDE project

1. If you are using the FreeRTOS+TCP TCP/IP stack, add `<freertos>/libraries/abstractions/secure_sockets/freertos_plus_tcp/iot_secure_sockets.c` to the `aws_tests` IDE project.

If you are using the lwIP TCP/IP stack, add `<freertos>/libraries/abstractions/secure_sockets/lwip/iot_secure_sockets.c` to the `aws_tests` IDE project.

If you are using your own TCP/IP port, add `<freertos>/vendors/<vendor>/boards/<board>/ports/secure_sockets/iot_secure_sockets.c` to the `aws_tests` IDE project.

2. Add `secure_sockets/test/aws_test_tcp.c` to the `aws_tests` IDE project.

Configuring the CMakeLists.txt File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

See below for an example portable layer target definition for the Secure Sockets library.

```
# Secure sockets
afr_mcu_port(secure_sockets)
# Link to AFR::secure_sockets_freertos_tcp if you want use default implementation based on
# FreeRTOS-Plus-TCP.
target_link_libraries(
    AFR::pkcs11::mcu_port
    INTERFACE AFR::secure_sockets_freertos_tcp
)
# Or provide your own implementation.
target_sources(
    AFR::secure_sockets::mcu_port
    INTERFACE "$path/iot_secure_sockets.c"
)
```

Setting Up Your Local Testing Environment

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the Secure Sockets tests

1. Open `<freertos>/libraries/freertos_plus/standard/utils/src/iot_system_init.c`, and in the function `SYSTEM_Init()`, comment out the lines that call `BUFFERPOOL_Init()` and `MQTT_AGENT_Init()`, if you have not done so already. Bufferpool and the MQTT agent are not used in this library's porting tests. When you reach the [Configuring the MQTT Library for Testing \(p. 67\)](#) section, you will be instructed to uncomment these initialization function calls for testing the MQTT library.

Make sure that the line that calls `SOCKETS_Init()` is uncommented.

2. Start an echo server.

If you have not ported the TLS library to your platform, you can only test your Secure Sockets port using an unsecured echo server (`<freertos>/tools/echo_server/echo_server.go`). For instructions on setting up and running an unsecured echo server, see [Setting Up the Echo Server \(Without TLS\) \(p. 44\)](#).

3. In `aws_test_tcp.h`, set the IP address to the correct values for your server. For example, if your server's IP address is `192.168.2.6`, set the following values in `aws_test_tcp.h`:

Macro	Value
<code>tcptestECHO_SERVER_ADDR0</code>	192
<code>tcptestECHO_SERVER_ADDR1</code>	168
<code>tcptestECHO_SERVER_ADDR2</code>	2
<code>tcptestECHO_SERVER_ADDR3</code>	6

4. Open `aws_test_tcp.h`, and set the `tcptestSECURE_SERVER` macro to 0 to run the Secure Sockets tests without TLS.

5. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner.config.h`, and set the `testrunnerFULL_TCP_ENABLED` macro to 1 to enable the sockets tests.
6. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/application_code/main.c`, and delete the `#if 0` and `#endif` compiler directives in the `vApplicationIPNetworkEventHook (void)` definition to enable the testing task.

Note

This change is required to port the remaining libraries.

Important

For qualification, you must pass the Secure Sockets tests with TLS. After you port the TLS library, rerun the Secure Sockets tests with TLS tests enabled, using a TLS-capable echo server. To port the TLS library, see [Porting the TLS Library \(p. 49\)](#).

To set up testing for Secure Sockets after porting the TLS library

1. Start a secure echo server.

For information, see [Setting Up the TLS Echo Server \(p. 44\)](#).

2. Set the IP address and port in `<freertos>/tests/include/aws_test_tcp.h` to correct values for your server. For example, if your server's IP address is 192.168.2.6, and the server is listening on 9000, set the following values in `<freertos>/tests/include/aws_test_tcp.h`:

Macro	Value
<code>tcptestECHO_SERVER_TLS_ADDR0</code>	192
<code>tcptestECHO_SERVER_TLS_ADDR1</code>	168
<code>tcptestECHO_SERVER_TLS_ADDR2</code>	2
<code>tcptestECHO_SERVER_TLS_ADDR3</code>	6
<code>tcptestECHO_PORT_TLS</code>	9000

3. Open `<freertos>/tests/include/aws_test_tcp.h`, and set the `tcptestSECURE_SERVER` macro to 1 to enable TLS tests.
4. Download a trusted root certificate. For information about accepted root certificates and download links, see [Server Authentication](#) in the AWS IoT Developer Guide. We recommend that you use Amazon Trust Services certificates.
5. In a browser window, open `<freertos>/tools/certificate_configuration/PEMfileToCString.html`.
6. Under **PEM Certificate or Key**, choose the root CA file that you downloaded.
7. Choose **Display formatted PEM string to be copied into `aws_clientcredential_keys.h`**, and then copy the certificate string.
8. Open `aws_test_tcp.h`, and paste the formatted certificate string into the definition for `tcptestECHO_HOST_ROOT_CA`.
9. Use the second set of OpenSSL commands in `<freertos>/tools/echo_server/readme-gencert.txt` to generate a client certificate and private key that is signed by the certificate authority. The certificate and key allow the custom echo server to trust the client certificate that your device presents during TLS authentication.
10. Format the certificate and key with the `<freertos>/tools/certificate_configuration/PEMfileToCString.html` formatting tool.

11. Before you build and run the test project on your device, open `aws_clientcredential_keys.h`, and copy the client certificate and private key, in PEM format, into the definitions for `keyCLIENT_CERTIFICATE_PEM` and `keyCLIENT_PRIVATE_KEY_PEM`.

Running the Tests

To execute the Secure Sockets tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console.

```
Starting Tests...
```

```
TEST(Full_TCP, SOCKETS_CloseInvalidParams) PASS
```

```
...
```

```
TEST(Full_TCP, SECURE_SOCKETS_NonBlockingConnect) PASS
```

```
TEST(Full_TCP, SECURE_SOCKETS_TwoSecureConnections) PASS
```

```
TEST(Full_TCP, SECURE_SOCKETS_SetSecureOptionsAfterConnect) PASS
```

```
-----
```

```
47 Tests 3 Failures 0 Ignored
```

```
FAIL
```

```
----All tests finished----
```

If all tests pass, then testing is complete.

Setting Up an Echo Server

Two simple echo servers, written in Go, are provided with FreeRTOS. One server uses TLS for secure communication, and the other is unsecured. The servers are located in the `<freertos>/tools/echo_server` folder. The following topics walk you through setting up the echo servers.

Topics

- [Setting Up the TLS Echo Server \(p. 44\)](#)
- [Setting Up the Echo Server \(Without TLS\) \(p. 44\)](#)

Setting Up the TLS Echo Server

A TLS echo server is defined in `<freertos>/tools/echo_server/tls_echo_server.go`. Follow these instructions to set up and run the TLS echo server.

Prerequisites

To run the TLS echo server, you need to install the following:

- Go.

You can download the latest version from golang.org.

- OpenSSL.

For a Linux source code download, see [OpenSSL.org](https://www.openssl.org). You can also use a package manager to install OpenSSL for Linux and macOS.

Setting Up the Server

1. Copy `<freertos>/tools/echo_server/tls_echo_server.go` to a directory of your choice (for example, `<echo_dir>`).
2. Under `<echo_dir>`, create a subfolder named `certs`.
3. Generate a TLS server self-signed certificate and private key. For OpenSSL commands to generate a self-signed server certificate and private key, see `<freertos>/tools/echo_server/readme-gencert.txt`.
4. Copy the self-signed certificate and private key `.pem` files to the `certs` directory.
5. Run the following command from the `<echo_dir>` directory to start the TLS server:

```
go run tls_echo_server.go
```

The server listens on port 9000 by default. To change this port, open `tls_echo_server.go`, and redefine the `sEchoPort` string to the port number that you want.

Setting Up the Echo Server (Without TLS)

An unsecured echo server is defined in `<freertos>/tools/echo_server/echo_server.go`. Follow these instructions to set up and run the echo server.

Prerequisites

To run the echo server, you need the latest version of Go. You can download the latest version from golang.org.

Setting Up the Server

1. Copy `<freertos>/tools/echo_server/echo_server.go` to a directory of your choice (for example, `<echo_dir>`).
2. Run the following command from the `<echo_dir>` directory to start the server:

```
go run echo_server.go
```

The server listens on port 9001 by default. To change this port, open `echo_server.go`, and redefine the `sUnSecureEchoPort` string to the port number that you want.

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the `device.json` file in the Device Tester configs folder.

After you finish porting the FreeRTOS Secure Sockets library to your device, you can start porting the PKCS #11 library. See [Porting the PKCS #11 Library \(p. 45\)](#) for instructions.

Porting the PKCS #11 Library

FreeRTOS uses the open standard PKCS #11 “CryptoKi” API as the abstraction layer for cryptographic operations, including:

- Signing and verifying.
- Storage and enumeration of X.509 certificates.
- Storage and management of cryptographic keys.

For more information, see [PKCS #11 Cryptographic Token Interface Base Specification](#).

Storing private keys in general-purpose flash memory can be convenient in evaluation and rapid prototyping scenarios. In production scenarios, to reduce the threats of data theft and device duplication, we recommend that you use dedicated cryptographic hardware. Cryptographic hardware includes components with features that prevent cryptographic secret keys from being exported. To use dedicated cryptographic hardware with FreeRTOS, you need to port the PKCS #11 API to the hardware. For information about the FreeRTOS PKCS #11 library, see [FreeRTOS PKCS #11 Library](#) in the FreeRTOS User Guide.

Prerequisites

To port the PKCS #11 library, you need the following:

- An IDE project or `CMakeLists.txt` list file that includes vendor-supplied drivers that are suitable for sensitive data.

For information about setting up a test project, see [Setting Up Your FreeRTOS Source Code for Porting \(p. 9\)](#).

- A validated configuration of the FreeRTOS kernel.

For information about configuring the FreeRTOS kernel for your platform, see [Configuring a FreeRTOS Kernel Port \(p. 28\)](#).

Porting

To port the PKCS #11 library

1. Port the PKCS #11 API functions.

The PKCS #11 API is dependent on the implementation of cryptographic primitives, such as SHA256 hashing and Elliptic Curve Digital Signature Algorithm (ECDSA) signing.

The FreeRTOS implementation of PKCS #11 uses the cryptographic primitives implemented in the mbedTLS library. FreeRTOS includes a port for mbedTLS. If your target hardware offloads crypto to a separate module, or if you want to use a software implementation of the cryptographic primitives other than mbedTLS, you need to modify the existing PKCS #11 port.

2. Port the PKCS # 11 Platform Abstraction Layer (PAL) for device-specific certificate and key storage.

If you decide to use the FreeRTOS implementation of PKCS #11, little customization is required to read and write cryptographic objects to non-volatile memory (NVM), such as onboard flash memory.

Cryptographic objects should be stored in a section of NVM that is not initialized and is not erased on device reprogramming. Users of the PKCS #11 library should be able to provision devices with credentials, and then reprogram the device with a new application that accesses these credentials through the PKCS #11 interface.

PKCS #11 PAL ports must provide a location to store:

- The device client certificate.
- The device client private key.
- The device client public key.
- A trusted root CA.
- A code-verification public key (or a certificate that contains the code-verification public key) for secure bootloader and over-the-air (OTA) updates.
- A Just-In-Time provisioning certificate.

`<freertos>/vendors/<vendor>/boards/<board>/ports/pkcs11/iot_pkcs11_pal.c` contains empty definitions for the PAL functions. You must provide ports for, at minimum, the functions listed in this table:

Function	Description
PKCS11_PAL_SaveObject	Writes data to non-volatile storage.
PKCS11_PAL_FindObject	Uses a PKCS #11 CKA_LABEL to search for a corresponding PKCS #11 object in non-volatile storage, and returns that object's handle, if it exists.
PKCS11_PAL_GetObjectValue	Retrieves the value of an object, given the handle.
PKCS11_PAL_GetObjectValueCleanup	Cleanup for the PKCS11_PAL_GetObjectValue call. Can be used to free memory allocated in a PKCS11_PAL_GetObjectValue call.

3. Add support for a cryptographically random entropy source to your port:
 - If your ports use the mbedTLS library for underlying cryptographic and TLS support, and your device has a true random number generator (TRNG):
 1. Implement the `mbedtls_hardware_poll()` function to seed the deterministic random bit generator (DRBG) that mbedTLS uses to produce a cryptographically random bit stream. The `mbedtls_hardware_poll()` function is located in `<freertos>/vendors/<vendor>/boards/<board>/ports/pkcs11/iot_pkcs11_pal.c`.
 - If your ports use the mbedTLS library for underlying cryptographic and TLS support, but your device does not have a TRNG:

1. Make a copy of `<freertos>/libraries/3rdparty/mbedtls/include/mbedtls/config.h`, and in that copy, uncomment `MBEDTLS_ENTROPY_NV_SEED`, and comment out `MBEDTLS_ENTROPY_HARDWARE_ALT`.

Save the modified version of `config.h` to `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/config.h`. Do not overwrite the original file.

2. Implement the functions `mbedtls_nv_seed_poll()`, `nv_seed_read_func()`, and `nv_seed_write_func()`.

For information about implementing these functions, see the comments in the [mbedtls/include/mbedtls/entropy_poll.h](#) and [mbedtls/include/mbedtls/config.h](#) mbedTLS header files.

Important

A seed file with an NIST-approved entropy source must be supplied to the device at manufacturing time.

Note

If you are interested in the FreeRTOS Qualification Program, please read our requirements for [RNG](#).

For more information about NIST-approved DRBGs and entropy sources, see the following NIST publications:

- [Recommendation for Random Number Generation Using Deterministic Random Bit Generators](#)
- [Recommendation for Random Bit Generator \(RBG\) Constructions](#)

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the PKCS #11 library in the IDE project

1. Add the source file `<freertos>/vendors/<vendor>/boards/<board>/ports/pkcs11/iot_pkcs11_pal.c` to the `aws_tests` IDE project.
2. Add all of the files in the `<freertos>/libraries/abstractions/pkcs11` directory and its subdirectories to the `aws_tests` IDE project.
3. Add all of the files in the `<freertos>/libraries/freertos_plus/standard/pkcs11` directory and its subdirectories to the `aws_tests` IDE project. These files implement wrappers for commonly grouped PKCS #11 function sets.
4. Add the source file `<freertos>/libraries/freertos_plus/standard/crypto/src/aws_crypto.c` to the `aws_tests` IDE project. This file implements the CRYPTO abstraction wrapper for mbedTLS.
5. Add all of the source and header files from `<freertos>/libraries/3rdparty/mbedtls` and its subdirectories to the `aws_tests` IDE project.

6. Add `<freertos>/libraries/3rdparty/mbedtls/include` and `<freertos>/libraries/abstractions/pkcs11` to the compiler's include path.

Configuring the CMakeLists.txt File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in CMakeLists.txt, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The CMakeLists.txt template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

See below for an example portable layer target definition for the PKCS #11 library that uses the mbedtls-based software implementation of PKCS #11 and supplies a port-specific PKCS #11 PAL file.

```
# PKCS11
afr_mcu_port(pkcs11_implementation DEPENDS AFR::pkcs11_mbedtls)
target_sources(
    AFR::pkcs11_implementation::mcu_port
    INTERFACE
    "${afr_ports_dir}/pkcs11/iot_pkcs11_pal.c"
)
```

Setting Up Your Local Testing Environment

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the PKCS #11 tests

1. Open `<freertos>/libraries/freertos_plus/standard/utils/src/iot_system_init.c`, and in the function `SYSTEM_Init()`, comment out the lines that call `BUFFERPOOL_Init()` and `MQTT_AGENT_Init()`, if you have not done so already. Bufferpool and the MQTT agent are not used in this library's porting tests. When you reach the [Configuring the MQTT Library for Testing \(p. 67\)](#) section, you will be instructed to uncomment these initialization function calls for testing the MQTT library.

Only uncomment calls to `SOCKETS_Init()` if you have ported the Secure Sockets library.

2. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_PKCS11_ENABLED` macro to 1 to enable the PKCS #11 test.

Running the Tests

To execute the PKCS #11 tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console.

```
-----STARTING TESTS-----  
TEST(Full_PKCS11_StartFinish, AFQP_StartFinish_FirstTest) PASS  
TEST(Full_PKCS11_StartFinish, AFQP_GetFunctionList) PASS  
TEST(Full_PKCS11_StartFinish, AFQP_InitializeFinalize) PASS  
TEST(Full_PKCS11_StartFinish, AFQP_GetSlotList) PASS  
TEST(Full_PKCS11_StartFinish, AFQP_OpenSessionCloseSession) PASS  
TEST(Full_PKCS11_NoObject, AFQP_Digest) PASS  
TEST(Full_PKCS11_NoObject, AFQP_Digest_ErrorConditions) PASS  
TEST(Full_PKCS11_NoObject, AFQP_GenerateRandom) PASS
```

...

```
TEST(Full_PKCS11_EC, AFQP_GenerateKeyPair) PASS  
15 9494 [TestRunner] Write certificate...  
TEST(Full_PKCS11_EC, AFQP_GetAttributeValueMultiThread) PASS  
TEST(Full_PKCS11_EC, AFQP_FindObjectMultiThread) PASS  
TEST(Full_PKCS11_EC, AFQP_SignVerifyMultiThread) PASS  
16 46939 [TestRunner] Write certificate...  
17 46971 [TestRunner] Write certificate...  
18 46971 [TestRunner] Warning: Device does not have memory allocated for just in time provisioning (JITP) certificate.  
Certificate from aws_clientcredential_keys.h will be used for JITP.  
19 46971 [TestRunner] Device credential provisioning succeeded.  
  
-----  
23 Tests 0 Failures 0 Ignored  
OK  
-----ALL TESTS FINISHED-----
```

Testing is complete when all tests pass.

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the device.json file in the Device Tester configs folder.

After you finish porting the FreeRTOS PKCS #11 library to your device, you can start porting the TLS library. See [Porting the TLS Library \(p. 49\)](#) for instructions.

Porting the TLS Library

For Transport Layer Security (TLS) authentication, FreeRTOS uses either mbedTLS or an off-chip TLS implementation, such as those found on some network co-processors. FreeRTOS includes a port of mbedTLS. If you use mbedTLS for TLS, TLS porting is not required. To allow different TLS implementations, third-party TLS libraries are accessed through a TLS abstraction layer.

Note

No matter which TLS implementation is used by your device's port of FreeRTOS, the port must pass the qualification tests for TLS. Qualification is based on results from AWS IoT Device Tester.

To prepare your platform for testing TLS, you need to configure your device in the AWS Cloud, and you need certificate and key provisioning on the device.

Prerequisites

To port the FreeRTOS TLS library, you need the following:

- A port of the FreeRTOS Secure Sockets library.

For information about porting the Secure Sockets library to your platform, see [Porting the Secure Sockets Library \(p. 39\)](#).

- A port of the FreeRTOS PKCS #11 library.

For information about porting the PKCS #11 library to your platform, see [Porting the PKCS #11 Library \(p. 45\)](#).

- An AWS account.

For information about setting up an AWS account, see [How do I create and activate a new Amazon Web Services account?](#) on the AWS Knowledge Center.

- OpenSSL.

You can download a version of OpenSSL for Windows from [Shining Light](#). For a Linux source code download, see [OpenSSL.org](#). You can also use a package manager to install OpenSSL for Linux and macOS.

Porting

If your target hardware offloads TLS functionality to a separate network chip, you need to implement the TLS abstraction layer functions in the following table.

Function	Description
<code>TLS_Init</code>	Initialize the TLS context.
<code>TLS_Connect</code>	Negotiate TLS and connect to the server.
<code>TLS_Recv</code>	Read the requested number of bytes from the TLS connection.
<code>TLS_Send</code>	Write the requested number of bytes to the TLS connection.
<code>TLS_Cleanup</code>	Free resources consumed by the TLS context.

`iot_tls.h` contains the information required to implement these functions. Save the file in which you implement the functions as `iot_tls.c`.

Connecting Your Device to AWS IoT

Your device must be registered with AWS IoT to communicate with the AWS Cloud. To register your board with AWS IoT, you need the following:

An AWS IoT policy

The AWS IoT policy grants your device permissions to access AWS IoT resources. It is stored in the AWS Cloud.

An AWS IoT thing

An AWS IoT thing allows you to manage your devices in AWS IoT. It is stored in the AWS Cloud.

A private key and X.509 certificate

The private key and certificate allow your device to authenticate with AWS IoT.

Follow these procedures to create a policy, thing, and key and certificate.

To create an AWS IoT policy

1. Browse to the [AWS IoT console](#).
2. In the navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**.
3. Enter a name to identify your policy.
4. In the **Add statements** section, choose **Advanced mode**. Copy and paste the following JSON into the policy editor window:

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

Important

This policy grants all AWS IoT resources access to all AWS IoT actions. This policy is convenient for development and testing purposes, but it is not recommended for production.

5. Choose **Create**.

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

1. Browse to the [AWS IoT console](#).
2. In the navigation pane, choose **Manage**, and then choose **Things**.
3. If you do not have any things registered in your account, the **You don't have any things yet** page is displayed. If you see this page, choose **Register a thing**. Otherwise, choose **Create**.
4. On the **Creating AWS IoT things** page, choose **Create a single thing**.
5. On the **Add your device to the thing registry** page, enter 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.
8. Choose **Activate** to activate your certificate. Certificates must be activated prior to use.
9. Choose **Attach a policy** to attach a policy to your certificate that grants your device access to AWS IoT operations.
10. Choose the policy you just created, and then choose **Register thing**.

After you obtain your certificates and keys from the AWS IoT console, you need to configure the `freertos/tests/include/aws_clientcredential.h` header file so your device can connect to AWS IoT.

To configure `freertos/tests/include/aws_clientcredential.h`

1. Browse to the [AWS IoT console](#).
2. In the navigation pane, choose **Settings**.

Your AWS IoT endpoint is displayed in **Endpoint**. It should look like `123456789012-ats.iot.us-east-1.amazonaws.com`. Make a note of this endpoint.

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

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

4. On the computer where you build the FreeRTOS source code, open the `freertos/test/include/aws_clientcredential.h` file in your IDE and specify values for the following constants:
 - `static const char clientcredentialMQTT_BROKER_ENDPOINT[] = "Your AWS IoT endpoint";`
 - `#define clientcredentialIOT_THING_NAME "The AWS IoT thing name of your board"`

Setting Up Certificates and Keys for the TLS Tests

TLS_ConnectRSA()

This section provides instructions on setting up certificates and keys for testing your TLS port.

For RSA device authentication, you can use the private key and the certificate that you downloaded from the AWS IoT console when you registered your device.

Note

After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

FreeRTOS is a C language project. You must format certificates and keys before you add them to the `freertos/tests/include/aws_clientcredential_keys.h` header file.

To format the certificate and key for `freertos/tests/include/aws_clientcredential_keys.h`

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

Note

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

TLS_ConnectEC()

For Elliptic Curve Digital Signature Algorithm (ECDSA) authentication, you need to generate a private key, a certificate signing request (CSR), and a certificate. You can use OpenSSL to generate the private key and CSR, and you can use the CSR to generate the certificate in the AWS IoT console.

To generate a private key and a CSR

1. Use the following command to create a private key file named `<p256_privatekey>.pem` in the current working directory:

```
openssl ecparam -name prime256v1 -genkey -noout -out <p256_privatekey>.pem
```


2. Use the following command to create a CSR file named `<csr>.csr` in the current working directory.

```
openssl req -new -key p256_privatekey.pem -out <csr>.csr
```

To create a certificate in the AWS IoT console with a CSR

1. Open the [AWS IoT console](#).
2. In the navigation pane, choose **Secure**, choose **Certificates**, and then choose **Create**.
3. Choose **Create with CSR**, and then find and upload the `<csr>.csr` file that you created with OpenSSL.
4. Choose **Activate** to activate the certificate, and then choose **Download** to download the certificate as a `.cert.pem` file.
5. Choose **Attach a policy**, and then find and select the AWS IoT policy that you created and attached to your RSA certificate in the [Connecting Your Device to AWS IoT \(p. 50\)](#) instructions, and choose **Done**.
6. Attach the certificate to the AWS IoT thing that you created when you registered your device.
7. From the **Certificates** page, find and select the certificate that you just created. From the upper right of the page, choose **Actions**, and then choose **Attach thing**.
8. Find and select the thing that you created for your device, and then choose **Attach**.

You must format the certificate and private key for your device before you add them to the `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h` header file.

To format the certificate and key for `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`

1. In a browser window, open `freertos/tools/certificate_configuration/PEMfileToCString.html`.
2. Under **PEM Certificate or Key**, choose the `ID-certificate.pem.crt` that you downloaded from the AWS IoT console.
3. Choose **Display formatted PEM string to be copied into aws_clientcredential_keys.h**, and then copy the certificate string.
4. Open `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`, and paste the formatted certificate string into the definition for `tlstestCLIENT_CERTIFICATE_PEM_EC`.

Note

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

5. Follow the same steps to get the formatted string for the private key file that you created using OpenSSL (`<p256_privatekey>.pem`). Copy and paste the formatted private key string into the definition for `tlstestCLIENT_PRIVATE_KEY_PEM_EC` in `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`.

In `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`, define the `tlstestMQTT_BROKER_ENDPOINT_EC` with the same AWS IoT MQTT broker endpoint address that you used in [Connecting Your Device to AWS IoT \(p. 50\)](#).

TLS_ConnectMalformedCert()

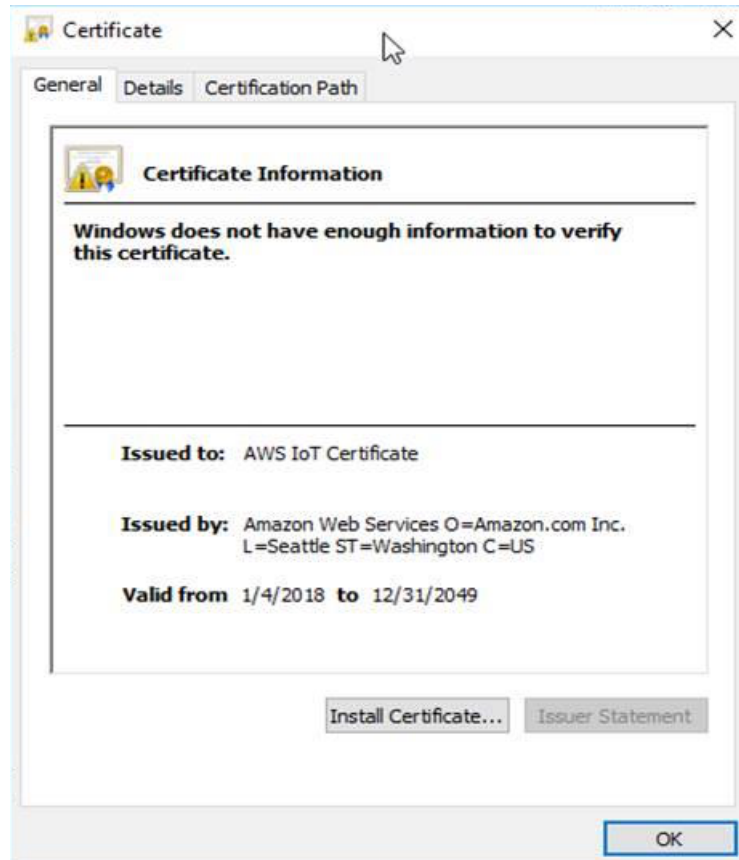
This test verifies that you can use a malformed certificate to authenticate with the server. Random modification of a certificate is likely to be rejected by X.509 certificate verification before the connection

request is sent out. To set up a malformed certificate, we suggest that you modify the issuer of the certificate.

To modify the issuer of a certificate

1. Take the valid client certificate that you have been using, `ID-certificate.pem.crt`.

In the Windows Certificate Manager, the certificate properties appear as follows:



2. Using the following command, convert the certificate from PEM to DER:

```
openssl x509 -outform der -in ID-certificate.pem.crt -out ID-certificate.der.crt
```

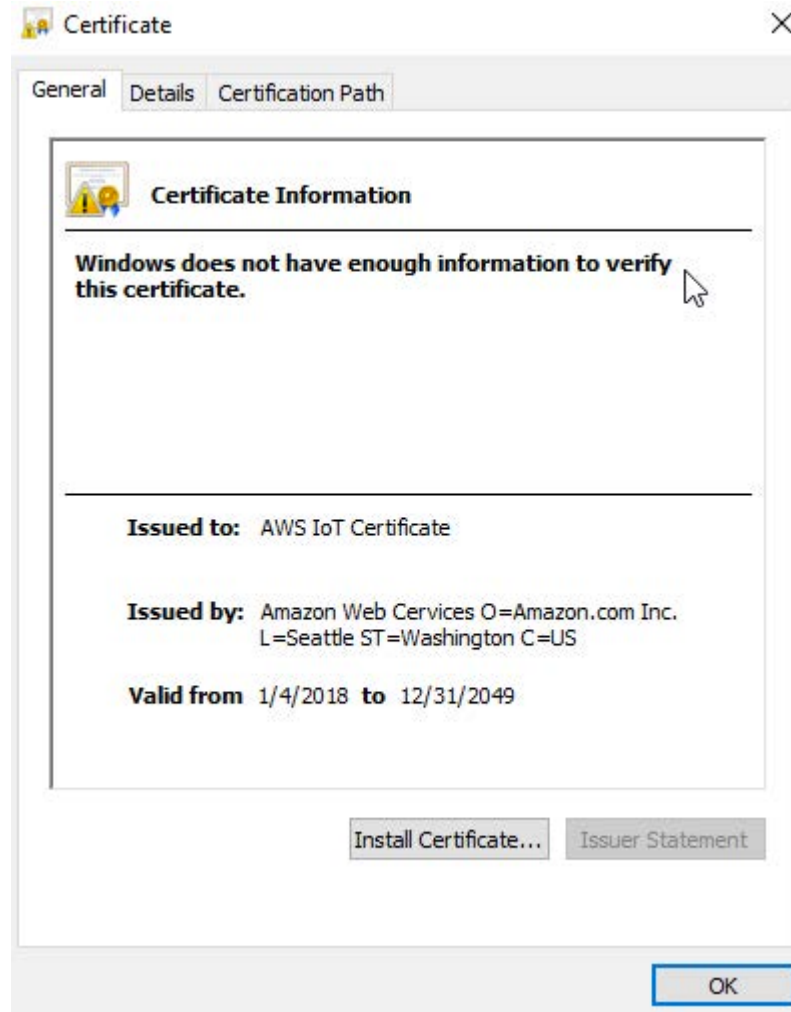
3. Open the DER certificate, and search for the following hexadecimal sequence:

```
41 6d 61 7a 6f 6e 20 57 65 62 20 53 65 72 76 69 63 65 73
```

This sequence, translated to plain text, reads "Amazon Web Services."

4. Change the 53 to 43, so that the sequence becomes "Amazon Web Services" in plain text, and save the file.

In the Windows Certificate Manager, the certificate properties now appear as follows:



5. Use the following command to convert the certificate back to PEM:

```
openssl x509 -inform der -in ID-certificate.der.crt -out ID-cert-modified.pem.crt
```

You must format the malformed certificate for your device before you add it to the `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h` header file.

To format the certificate for `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`

1. In a browser window, open `freertos/tools/certificate_configuration/PEMfileToCString.html`.
2. Under **PEM Certificate or Key**, choose the `ID-certificate.pem.crt` that you created and then modified.
3. Choose **Display formatted PEM string to be copied into `aws_clientcredential_keys.h`**, and then copy the certificate string.
4. Open `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`, and paste the formatted certificate string into the definition for `tlstestCLIENT_CERTIFICATE_PEM_MALFORMED`.

Note

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

TLS_ConnectBYOCCredentials()

You can use your own certificate for authentication. To create and register a certificate with a valid rootCA/CA chain, follow the instructions in [Creating a BYOC \(ECDSA\) \(p. 57\)](#). After you create the certificate, you need to attach some policies to your device certificate, and then you need to attach your device's thing to the certificate.

To attach a policy to your device certificate

1. Open the [AWS IoT console](#).
2. In the navigation pane, choose **Secure**, choose **Certificates**, and then choose the device certificate that you created and registered in [Creating a BYOC \(ECDSA\) \(p. 57\)](#).
3. Choose **Actions**, and then choose **Attach policy**.
4. Find and choose the AWS IoT policy that you created and attached to your RSA certificate in the [Connecting Your Device to AWS IoT \(p. 50\)](#) instructions, and then choose **Attach**.

To attach a thing to your device certificate

1. From the **Certificates** page, find and choose the same device certificate, choose **Actions**, and then choose **Attach thing**.
2. Find and choose the thing that you created for your device, and then choose **Attach**.

To format the certificate for `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`

1. In a browser window, open `freertos/tools/certificate_configuration/PEMfileToCString.html`.
2. Under **PEM Certificate or Key**, choose the `ID-certificate.pem.crt` that you created and then modified.
3. Choose **Display formatted PEM string to be copied into aws_clientcredential_keys.h**, and then copy the certificate string.
4. Open `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`, and paste the formatted certificate string into the definition for `tlstestCLIENT_BYOC_CERTIFICATE_PEM`.

Note

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

5. Follow the same steps to get the formatted string for the private key file that you created. Copy and paste the formatted private key string into the definition for `tlstestCLIENT_BYOC_PRIVATE_KEY_PEM` in `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`.

TLS_ConnectUntrustedCert()

You can use your own certificate for authentication, without registering your certificate with AWS IoT. To create a certificate with a valid rootCA/CA chain, follow the instructions in [Creating a BYOC \(ECDSA\) \(p. 57\)](#), but skip the final instructions for registering your device with AWS IoT.

To format the certificate for `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`

1. In a browser window, open `freertos/tools/certificate_configuration/PEMfileToCString.html`.
2. Under **PEM Certificate or Key**, choose the `ID-certificate.pem.crt` that you created and then modified.
3. Choose **Display formatted PEM string to be copied into `aws_clientcredential_keys.h`**, and then copy the certificate string.
4. Open `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`, and paste the formatted certificate string into the definition for `tlstestCLIENT_UNTRUSTED_CERTIFICATE_PEM`.

Note

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

5. Follow the same steps to get the formatted string for the private key file that you created. Copy and paste the formatted private key string into the definition for `tlstestCLIENT_UNTRUSTED_PRIVATE_KEY_PEM` in `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`.

Creating a BYOC (ECDSA)

In these procedures, you use the AWS IoT console, the AWS Command Line Interface, and OpenSSL to create and register certificates and keys for a device on the AWS Cloud. Make sure that you have installed and configured the AWS CLI on your machine before you run the AWS CLI commands.

Note

When you create CA certificates, use valid, consistent values for the Distinguished Name (DN) fields, when prompted. For the Common Name field, you can use any value, unless otherwise instructed.

To generate a root CA

1. Use the following command to generate a root CA private key:

```
openssl ecparam -name prime256v1 -genkey -noout -out rootCA.key
```

2. Use the following command to generate a root CA certificate:

```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.crt
```

To generate an intermediate CA

1. Create required files:

```
touch index.txt
```

```
echo 1000 > serial
```

2. Save the [ca.config](#) (p. 59) file in the current working directory.
3. Use the following command to generate the intermediate CA's private key:

```
openssl ecparam -name prime256v1 -genkey -noout -out intermediateCA.key
```

4. Use the following command to generate the intermediate CA's CSR:

```
openssl req -new -sha256 -key intermediateCA.key -out intermediateCA.csr
```

5. Use the following command to sign the intermediate CA's CSR with the root CA:

```
openssl ca -config ca.config -notext -cert rootCA.crt -keyfile rootCA.key -days 500 -in  
intermediateCA.csr -out intermediateCA.crt
```

To generate a device certificate

Note

An ECDSA certificate is used here as an example.

1. Use the following command to generate a private key:

```
openssl ecparam -name prime256v1 -genkey -noout -out deviceCert.key
```

2. Use the following command to generate a CSR for a device certificate:

```
openssl req -new -key deviceCert.key -out deviceCert.csr
```

3. Use the following command to sign the device certificate with the intermediate CA:

```
openssl x509 -req -in deviceCert.csr -CA intermediateCA.crt -CAkey intermediateCA.key -  
CAcreateserial -out deviceCert.crt -days 500 -sha256
```

To register both CA certificates

1. Use the following AWS CLI command to get the registration code:

```
aws iot get-registration-code
```

2. Use the following command to generate a private key for verification certificates:

```
openssl ecparam -name prime256v1 -genkey -noout -out verificationCert.key
```

3. Use the following command to create CSR for verification certificates:

```
openssl req -new -key verificationCert.key -out verificationCert.csr
```

When prompted, for Common Name, enter the registration code that you obtained in the first step.

4. Use the following command to sign a verification certificate using the root CA:

```
openssl x509 -req -in verificationCert.csr -CA rootCA.crt -CAkey rootCA.key -  
CAcreateserial -out rootCAverificationCert.crt -days 500 -sha256
```

5. Use the following command to sign a verification certificate using the intermediate CA:

```
openssl x509 -req -in verificationCert.csr -CA intermediateCA.crt -CAkey  
intermediateCA.key -CAcreateserial -out intermediateCAverificationCert.crt -days 500 -  
sha256
```

6. Use the following AWS CLI commands to register both CA certificates with AWS IoT:

```
aws iot register-ca-certificate --ca-certificate file://rootCA.crt --verification-cert  
file://rootCAverificationCert.crt
```

```
aws iot register-ca-certificate --ca-certificate file://intermediateCA.crt --  
verification-cert file://intermediateCAverificationCert.crt
```

7. Use the following AWS CLI command to activate both CA certificates:

```
aws iot update-ca-certificate --certificate-id ID --new-status ACTIVE
```

Where **ID** is the certificate ID of one of the certificates.

To register the device certificate

1. Use the following AWS CLI command to register the device certificate with AWS IoT:

```
aws iot register-certificate --certificate-pem file://deviceCert.crt --ca-certificate-  
pem file://intermediateCA.crt
```

2. Use the following AWS CLI command to activate the device certificate:

```
aws iot update-certificate --certificate-id ID --new-status ACTIVE
```

Where **ID** is the certificate ID of the certificate.

ca.config

Save the following text to a file named `ca.config` in your current working directory.

This file is a modified version of the [openssl.cnf](#) OpenSSL example configuration file.

```
#  
# OpenSSL example configuration file.  
# This is mostly being used for generation of certificate requests.  
#  
  
# This definition stops the following lines choking if HOME isn't  
# defined.  
HOME = .  
RANDFILE = $ENV::HOME/.rnd  
  
# Extra OBJECT IDENTIFIER info:  
#oid_file = $ENV::HOME/.oid  
oid_section = new_oids  
  
# To use this configuration file with the "-extfile" option of the  
# "openssl x509" utility, name here the section containing the  
# X.509v3 extensions to use:
```

```
# extensions =
# (Alternatively, use a configuration file that has only
# X.509v3 extensions in its main [= default] section.)

[ new_oids ]

# We can add new OIDs in here for use by 'ca', 'req' and 'ts'.
# Add a simple OID like this:
# testoid1=1.2.3.4
# Or use config file substitution like this:
# testoid2=${testoid1}.5.6

# Policies used by the TSA examples.
tsa_policy1 = 1.2.3.4.1
tsa_policy2 = 1.2.3.4.5.6
tsa_policy3 = 1.2.3.4.5.7

#####
[ ca ]
default_ca = CA_default # The default ca section

#####
[ CA_default ]

dir = . # Where everything is kept
certs = $dir # Where the issued certs are kept
crl_dir = $dir # Where the issued crl are kept
database = $dir/index.txt # database index file.
#unique_subject = no # Set to 'no' to allow creation of
# several certificates with same subject.
new_certs_dir = $dir # default place for new certs.

certificate = $dir/cacert.pem # The CA certificate
serial = $dir/serial # The current serial number
crlnumber = $dir/crlnumber # the current crl number
# must be commented out to leave a V1 CRL
crl = $dir/crl.pem # The current CRL
private_key = $dir/private/cakey.pem # The private key
RANDFILE = $dir/private/.rand # private random number file

x509_extensions = usr_cert # The extensions to add to the cert

# Comment out the following two lines for the "traditional"
# (and highly broken) format.
name_opt = ca_default # Subject Name options
cert_opt = ca_default # Certificate field options

# Extension copying option: use with caution.
# copy_extensions = copy

# Extensions to add to a CRL. Note: Netscape communicator chokes on V2 CRLs
# so this is commented out by default to leave a V1 CRL.
# crlnumber must also be commented out to leave a V1 CRL.
# crl_extensions = crl_ext

default_days = 365 # how long to certify for
default_crl_days = 30 # how long before next CRL
default_md = default # use public key default MD
preserve = no # keep passed DN ordering

# A few difference way of specifying how similar the request should look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy = policy_match
```



```
# For the CA policy
[ policy_match ]
countryName      = match
stateOrProvinceName  = match
organizationName   = match
organizationalUnitName = optional
commonName        = supplied
emailAddress       = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName      = optional
stateOrProvinceName  = optional
localityName     = optional
organizationName  = optional
organizationalUnitName = optional
commonName        = supplied
emailAddress       = optional

#####
[ req ]
default_bits      = 2048
default_keyfile    = privkey.pem
distinguished_name = req_distinguished_name
attributes         = req_attributes
x509_extensions   = v3_ca # The extensions to add to the self signed cert

# Passwords for private keys if not present they will be prompted for
# input_password = secret
# output_password = secret

# This sets a mask for permitted string types. There are several options.
# default: PrintableString, T61String, BMPString.
# pkix      : PrintableString, BMPString (PKIX recommendation before 2004)
# utf8only: only UTF8Strings (PKIX recommendation after 2004).
# nombstr : PrintableString, T61String (no BMPStrings or UTF8Strings).
# MASK:XXXX a literal mask value.
# WARNING: ancient versions of Netscape crash on BMPStrings or UTF8Strings.
string_mask = utf8only

# req_extensions = v3_req # The extensions to add to a certificate request

[ req_distinguished_name ]
countryName      = Country Name (2 letter code)
countryName_default  = AU
countryName_min    = 2
countryName_max    = 2

stateOrProvinceName      = State or Province Name (full name)
stateOrProvinceName_default  = Some-State

localityName              = Locality Name (eg, city)

0.organizationName        = Organization Name (eg, company)
0.organizationName_default  = Internet Widgits Pty Ltd

# we can do this but it is not needed normally :-)
#1.organizationName        = Second Organization Name (eg, company)
#1.organizationName_default  = World Wide Web Pty Ltd

organizationalUnitName     = Organizational Unit Name (eg, section)
#organizationalUnitName_default  =

commonName                  = Common Name (e.g. server FQDN or YOUR name)
```

```
commonName_max          = 64

emailAddress            = Email Address
emailAddress_max        = 64

# SET-ex3                = SET extension number 3

[ req_attributes ]
challengePassword       = A challenge password
challengePassword_min   = 4
challengePassword_max   = 20

unstructuredName        = An optional company name

[ usr_cert ]

# These extensions are added when 'ca' signs a request.

# This goes against PKIX guidelines but some CAs do it and some software
# requires this to avoid interpreting an end user certificate as a CA.

basicConstraints=CA:TRUE

# Here are some examples of the usage of nsCertType. If it is omitted
# the certificate can be used for anything *except* object signing.

# This is OK for an SSL server.
# nsCertType            = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature, keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment              = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl      = http://www.domain.dom/ca-crl.pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

# This is required for TSA certificates.
```

```
# extendedKeyUsage = critical,timeStamping

[ v3_req ]

# Extensions to add to a certificate request

basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment

[ v3_ca ]

# Extensions for a typical CA

# PKIX recommendation.

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid:always,issuer

# This is what PKIX recommends but some broken software chokes on critical
# extensions.

#basicConstraints = critical,CA:true
# So we do this instead.
basicConstraints = CA:true

# Key usage: this is typical for a CA certificate. However since it will
# prevent it being used as a test self-signed certificate it is best
# left out by default.
# keyUsage = cRLSign, keyCertSign

# Some might want this also
# nsCertType = sslCA, emailCA

# Include email address in subject alt name: another PKIX recommendation
# subjectAltName=email:copy
# Copy issuer details
# issuerAltName=issuer:copy

# DER hex encoding of an extension: beware experts only!
# obj=DER:02:03
# Where 'obj' is a standard or added object
# You can even override a supported extension:
# basicConstraints= critical, DER:30:03:01:01:FF

[ crl_ext ]

# CRL extensions.
# Only issuerAltName and authorityKeyIdentifier make any sense in a CRL.

# issuerAltName=issuer:copy
authorityKeyIdentifier=keyid:always

[ proxy_cert_ext ]
# These extensions should be added when creating a proxy certificate

# This goes against PKIX guidelines but some CAs do it and some software
# requires this to avoid interpreting an end user certificate as a CA.

basicConstraints=CA:FALSE
```

```
# Here are some examples of the usage of nsCertType. If it is omitted
# the certificate can be used for anything *except* object signing.

# This is OK for an SSL server.
# nsCertType          = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature, keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment          = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl      = http://www.domain.dom/ca-crl.pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

# This really needs to be in place for it to be a proxy certificate.
proxyCertInfo=critical,language:id-ppl-anyLanguage,pathlen:3,policy:foo

#####
[ tsa ]

default_tsa = tsa_config1  # the default TSA section

[ tsa_config1 ]

# These are used by the TSA reply generation only.
dir          = ./demoCA      # TSA root directory
serial       = $dir/tsaserial # The current serial number (mandatory)
crypto_device = builtin      # OpenSSL engine to use for signing
signer_cert  = $dir/tsacert.pem # The TSA signing certificate
                                # (optional)
certs        = $dir/cacert.pem # Certificate chain to include in reply
                                # (optional)
signer_key   = $dir/private/tsakey.pem # The TSA private key (optional)

default_policy = tsa_policy1  # Policy if request did not specify it
                                # (optional)
```

```
other_policies    = tsa_policy2, tsa_policy3    # acceptable policies (optional)
digests          = md5, sha1                  # Acceptable message digests (mandatory)
accuracy         = secs:1, millisecs:500, microseconds:100 # (optional)
clock_precision_digits = 0                    # number of digits after dot. (optional)
ordering         = yes                        # Is ordering defined for timestamps?
# (optional, default: no)
tsa_name         = yes                        # Must the TSA name be included in the reply?
# (optional, default: no)
ess_cert_id_chain = no                       # Must the ESS cert id chain be included?
# (optional, default: no)
```

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the TLS library in the IDE project

1. Add `iot_tls.c` to the `aws_tests` IDE project.
2. Add the source file `iot_test_tls.c` to the virtual folder `aws_tests/application_code/common_tests/tls`. This file includes the TLS tests.

Configuring the `CMakeLists.txt` File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The `CMakeLists.txt` template list file under `freertos/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

Setting Up Your Local Testing Environment

There are five separate tests for the TLS port, one for each type of authentication supported by the FreeRTOS TLS library:

- `TLS_ConnectRSA()`
- `TLS_ConnectEC()`
- `TLS_ConnectMalformedCert()`
- `TLS_ConnectBYOCCredentials()`
- `TLS_ConnectUntrustedCert()`

To run these tests, your board must use the MQTT protocol to communicate with the AWS Cloud. AWS IoT hosts an MQTT broker that sends and receives messages to and from connected devices at the edge. The AWS IoT MQTT broker accepts mutually authenticated TLS connections only.

Follow the instructions in [Connecting Your Device to AWS IoT \(p. 50\)](#) to connect your device to AWS IoT.

Each TLS test requires a different certificate/key combination, formatted and defined in either `freertos/tests/include/aws_clientcredential_keys.h` or `freertos/libraries/freertos_plus/standard/tls/test/iot_test_tls.h`.

Follow the instructions in [Setting Up Certificates and Keys for the TLS Tests \(p. 52\)](#) to obtain the certificates and keys that you need for testing.

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the TLS tests

1. To enable the TLS tests, open `freertos/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_TLS_ENABLED` macro to 1.
2. Open `freertos/libraries/freertos_plus/standard/utils/src/iot_system_init.c`, and in the function `SYSTEM_Init()`, comment out the lines that call `BUFFERPOOL_Init()` and `MQTT_AGENT_Init()`, if you have not done so already. Bufferpool and the MQTT agent are not used in this library's porting tests. When you reach the [Configuring the MQTT Library for Testing \(p. 67\)](#) section, you will be instructed to uncomment these initialization function calls for testing the MQTT library.

Make sure that the line that calls `SOCKETS_Init()` is uncommented.

Running the Tests

To execute the TLS tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console.

```
Starting Tests...

TEST(Full_TLS, AFQP_TLS_ConnectEC) PASS

TEST(Full_TLS, TLS_ConnectRSA) PASS

TEST(Full_TLS, TLS_ConnectMalformedCert) PASS

TEST(Full_TLS, TLS_ConnectUntrustedCert) PASS

TEST(Full_TLS, AFQP_TLS_ConnectBYOCCredentials) PASS

-----
5 Tests 0 Failures 0 Ignored
OK
-----ALL TESTS FINISHED-----
```

If all tests pass, then testing is complete.

Important

After you have ported the TLS library and tested your ports, you must run the Secure Socket tests that depend on TLS functionality. For more information, see [Testing \(p. 40\)](#) in the Secure Sockets porting documentation.

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the `device.json` file in the Device Tester configs folder.

After you finish porting the FreeRTOS TLS library to your device, you can start setting up the MQTT library for testing. See [Configuring the MQTT Library for Testing \(p. 67\)](#) for instructions.

Configuring the MQTT Library for Testing

Devices on the edge can use the MQTT protocol to communicate with the AWS Cloud. AWS IoT hosts an MQTT broker that sends and receives messages to and from connected devices at the edge.

The MQTT library implements the MQTT protocol for devices running FreeRTOS. The MQTT library does not need to be ported, but your device's test project must pass all MQTT tests for qualification. For more information, see [FreeRTOS MQTT Library](#) in the FreeRTOS User Guide.

Prerequisites

To set up the FreeRTOS MQTT library tests, you need the following:

- A port of the TLS library.

For information about porting the TLS library to your platform, see [Porting the TLS Library \(p. 49\)](#).

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

To set up the MQTT library in the IDE project

- Add all of the test source files in `<freertos>/libraries/c_sdk/standard/mqtt` and its subdirectories to the `aws_tests` IDE project.

Configuring the `CMakeLists.txt` File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

Setting Up Your Local Testing Environment

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the MQTT tests

- To enable the MQTT tests, open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_MQTTv4_ENABLED` macro to 1.

Running the Tests

To execute the MQTT tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console. If all tests pass, then testing is complete.

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the `device.json` file in the Device Tester `configs` folder.

After you finish setting up the FreeRTOS MQTT library for your device, you can start porting the OTA agent library. See [Porting the OTA Library \(p. 69\)](#) for instructions.

If your device does not support OTA functionality, you can start porting the Bluetooth Low Energy library. See [Porting the Bluetooth Low Energy Library \(p. 76\)](#) for instructions.

If your device does not support OTA and Bluetooth Low Energy functionality, then you are finished porting and can start the FreeRTOS qualification process. See the [FreeRTOS Qualification Guide](#) for more information.

Configuring the HTTPS Client Library for Testing

The HTTPS Client library implements the HTTPS/1.1 protocol over TLS for devices running FreeRTOS.

Prerequisites

To set up the FreeRTOS HTTPS Client library tests, you need the following:

- A port of the TLS library.

For information about porting the TLS library to your platform, see [Porting the TLS Library \(p. 49\)](#).

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

To set up the HTTPS Client library in the IDE project

- Add all of the test source files in `<freertos>/libraries/c_sdk/standard/https` and its subdirectories to the `aws_tests` IDE project.

Configuring the CMakeLists.txt File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in CMakeLists.txt, follow the instructions in [FreeRTOS Portable Layers](#) (p. 16).

The CMakeLists.txt template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

Setting Up Your Local Testing Environment

After you set up the library in the IDE project, you need to configure some other files for testing.

To configure the source and header files for the HTTPS Client tests

- To enable the HTTPS Client tests, open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_HTTPS_CLIENT_ENABLED` macro to 1.

Running the Tests

To execute the HTTPS Client tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console. If all tests pass, then testing is complete.

Porting the OTA Library

With FreeRTOS over-the-air (OTA) updates, you can do the following:

- Deploy new firmware images to a single device, a group of devices, or your entire fleet.
- Deploy firmware to devices as they are added to groups, are reset, or are reprovisioned.
- Verify the authenticity and integrity of new firmware after it has been deployed to devices.
- Monitor the progress of a deployment.
- Debug a failed deployment.
- Digitally sign firmware using Code Signing for AWS IoT.

For more information, see [FreeRTOS Over-the-Air Updates](#) in the *FreeRTOS User Guide*.

You can use the OTA agent library to integrate OTA functionality into your FreeRTOS applications. For more information, see [FreeRTOS OTA Agent Library](#) in the *FreeRTOS User Guide*.

FreeRTOS devices must enforce cryptographic code-signing verification on the OTA firmware images that they receive. We recommend the following algorithms:

- Elliptic-Curve Digital Signature Algorithm (ECDSA)
- NIST P256 curve
- SHA-256 hash

Note

A port of the FreeRTOS OTA update library is currently not required for qualification.

Prerequisites

To port the OTA agent library, you need the following:

- A port of the TLS library.

For information, see [Porting the TLS Library \(p. 49\)](#).

- A bootloader that can support OTA updates.

For more information about porting a bootloader demo application, see [Porting the Bootloader Demo \(p. 71\)](#).

Porting

`<freertos>/vendors/<vendor>/boards/<board>/ports/ota/aws_ota_pal.c` contains empty definitions of a set of platform abstraction layer (PAL) functions. Implement at least the set of functions listed in this table.

Function	Description
<code>prvPAL_Abort</code>	Aborts an OTA update.
<code>prvPAL_CreateFileForRx</code>	Creates a file to store the data chunks as they are received.
<code>prvPAL_CloseFile</code>	Closes the specified file. This might authenticate the file if storage that implements cryptographic protection is being used.
<code>prvPAL_WriteBlock</code>	Writes a block of data to the specified file at the given offset. On success, returns the number of bytes written. Otherwise, a negative error code.
<code>prvPAL_ActivateNewImage</code>	Activates or launches the new firmware image. For some ports, if the device is programmatically reset synchronously, this function might not return.
<code>prvPAL_SetPlatformImageState</code>	Does what is required by the platform to accept or reject the most recent OTA firmware image (or bundle). To determine how to implement this function, consult the documentation for your board (platform) details and architecture. .
<code>prvPAL_GetPlatformImageState</code>	Gets the state of the OTA update image.

Implement the functions in this table if your device has built-in support for them.

Function	Description
<code>prvPAL_CheckFileSignature</code>	Verifies the signature of the specified file.
<code>prvPAL_ReadAndAssumeCertificate</code>	Reads the specified signer certificate from the file system and returns it to the caller.

Make sure that you have a bootloader that can support OTA updates. For instructions on porting the bootloader demo application provided with FreeRTOS or creating your IoT device bootloader, see [IoT Device Bootloader \(p. 71\)](#).

IoT Device Bootloader

Porting the Bootloader Demo

FreeRTOS includes a demo bootloader application for the Microchip Curiosity PIC32MZEF platform. For more information, see [Demo Bootloader for the Microchip Curiosity PIC32MZEF](#) in the *FreeRTOS User Guide*. You can port this demo to other platforms.

If you choose not to port the demo to your platform, you can use your own bootloader application. [Threat Modeling for the IoT Device Bootloader \(p. 71\)](#) discusses threat modeling to help you write your own secure bootloader application.

Threat Modeling for the IoT Device Bootloader

Background

As a working definition, the embedded IoT devices referenced by this threat model are microcontroller-based products that interact with cloud services. They may be deployed in consumer, commercial, or industrial settings. IoT devices may gather data about a user, a patient, a machine, or an environment, and may control anything from light bulbs and door locks to factory machinery.

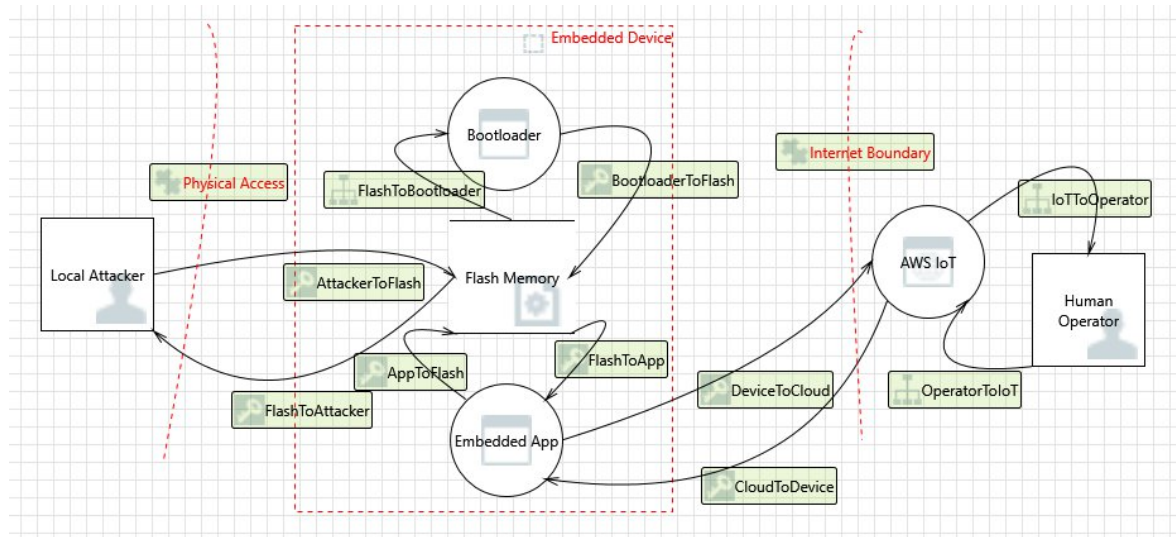
Threat modeling is an approach to security from the point of view of a hypothetical adversary. By considering the adversary's goals and methods, a list of threats is created. Threats are attacks against a resource or asset performed by an adversary. The list is prioritized and used to identify or create mitigations. When choosing mitigations, the cost of implementing and maintaining them should be balanced with the real security value they provide. There are multiple [threat model methodologies](#). Each is capable of supporting the development of a secure and successful IoT product.

FreeRTOS offers OTA ("over-the-air") software updates to IoT devices. The update facility combines cloud services with on-device software libraries and a partner-supplied bootloader. This threat model focuses specifically on threats against the bootloader.

Bootloader Use Cases

- Digitally sign and encrypt firmware before deployment.
- Deploy new firmware images to a single device, a group of devices, or an entire fleet.
- Verify the authenticity and integrity of new firmware after it's deployed to devices.
- Devices only run unmodified software from a trusted source.
- Devices are resilient to faulty software received through OTA.

Data Flow Diagram



Threats

Some attacks will have multiple mitigations; for example, a network man-in-the-middle intended to deliver a malicious firmware image is mitigated by verifying trust in both the certificate offered by the TLS server and the code-signer certificate of the new firmware image. To maximize the security of the bootloader, any non-bootloader mitigations are considered unreliable. The bootloader should have intrinsic mitigations for each attack. Having layered mitigations is known as defense-in-depth.

Threats:

- An attacker hijacks the device's connection to the server to deliver a malicious firmware image.

Mitigation example

- Upon boot, the bootloader verifies the cryptographic signature of the image using a known certificate. If the verification fails, the bootloader rolls back to the previous image.
- An attacker exploits a buffer overflow to introduce malicious behavior to the existing firmware image stored in flash.

Mitigation examples

- Upon boot, the bootloader verifies, as previously described. When verification fails with no previous image available, the bootloader halts.
- Upon boot, the bootloader verifies, as previously described. When verification fails with no previous image available, the bootloader enters a failsafe OTA-only mode.
- An attacker boots the device to a previously stored image, which is exploitable.

Mitigation examples

- Flash sectors storing the last image are erased upon successful installation and test of a new image.
- A fuse is burned with each successful upgrade, and each image refuses to run unless the correct number of fuses have been burned.
- An OTA update delivers a faulty or malicious image that bricks the device.

Mitigation example

- The bootloader starts a hardware watchdog timer that triggers rollback to the previous image.

- An attacker patches the bootloader to bypass image verification so the device will accept unsigned images.

Mitigation examples

- The bootloader is in ROM (read-only memory), and cannot be modified.
- The bootloader is in OTP (one-time-programmable memory), and cannot be modified.
- The bootloader is in the secure zone of ARM TrustZone, and cannot be modified.
- An attacker replaces the verification certificate so the device will accept malicious images.

Mitigation examples

- The certificate is in a cryptographic co-processor, and cannot be modified.
- The certificate is in ROM (or OTP, or secure zone), and cannot be modified.

Further Threat Modeling

This threat model considers only the bootloader. Further threat modeling could improve overall security. A recommended method is to list the adversary's goals, the assets targeted by those goals, and points of entry to the assets. A list of threats can be made by considering attacks on the points of entry to gain control of the assets. The following are lists of examples of goals, assets, and entry points for an IoT device. These lists are not exhaustive, and are intended to spur further thought.

Adversary's Goals

- Extort money
- Ruin reputations
- Falsify data
- Divert resources
- Remotely spy on a target
- Gain physical access to a site
- Wreak havoc
- Instill terror

Key Assets

- Private keys
- Client certificate
- CA root certificates
- Security credentials and tokens
- Customer's personally identifiable information
- Implementations of trade secrets
- Sensor data
- Cloud analytics data store
- Cloud infrastructure

Entry Points

- DHCP response
- DNS response
- MQTT over TLS

- HTTPS response
- OTA software image
- Other, as dictated by application, for example, USB
- Physical access to bus
- Decapped IC

Testing

If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the OTA library in the IDE project

1. Add the source file `<freertos>/vendors/<vendor>/boards/<board>/ports/ota/aws_ota_pal.c` to the `aws_tests` IDE project.
2. Add the following test source files to the `aws_tests` IDE project:
 - `<freertos>/libraries/freertos_plus/aws/ota/test/aws_test_ota_cbor.c`
 - `<freertos>/libraries/freertos_plus/aws/ota/test/aws_test_ota_agent.c`
 - `<freertos>/libraries/freertos_plus/aws/ota/test/aws_test_ota_pal.c`
 - `/demos/ota/aws_iot_ota_update_demo.c`

Configuring the CMakeLists.txt File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

The following is an example of a portable layer target definition for the OTA library.

```
# OTA
afr_mcu_port(ota)
target_sources(
    AFR::ota::mcu_port
    INTERFACE "path/aws_ota_pal.c"
)
```

There are two sets of tests for the OTA library port: [OTA Agent and OTA PAL Tests \(p. 75\)](#) and [OTA End-to-End Tests \(p. 76\)](#).

OTA Agent and OTA PAL Tests

Setting Up Your Local Testing Environment

To configure the source and header files for the OTA agent and OTA PAL tests

1. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_OTA_AGENT_ENABLED` and `testrunnerFULL_OTA_PAL_ENABLED` macros to 1 to enable the agent and PAL tests.
2. Choose a signing certificate for your device from `ota/test`. The certificate are used in OTA tests for verification.

Three types of signing certificates are available in the test code:

- RSA/SHA1
- RSA/SHA256
- ECDSA/SHA256

RSA/SHA1 and RSA/SHA256 are available for existing platforms only. ECDSA/SHA256 is recommended for OTA updates. If you have a different scheme, [contact the FreeRTOS engineering team](#).

Running the Tests

To execute the OTA agent and OTA PAL tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console.

```
-----STARTING TESTS-----  
  
TEST(Full_OTA_PAL, prvPAL_CloseFile_ValidSignature) PASS  
  
TEST(Full_OTA_PAL, prvPAL_CloseFile_InvalidSignatureBlockWritten) PASS  
  
TEST(Full_OTA_PAL, prvPAL_CloseFile_InvalidSignatureNoBlockWritten) PASS  
  
TEST(Full_OTA_PAL, prvPAL_CloseFile_NonexistingCodeSignerCertificate) PASS  
  
TEST(Full_OTA_PAL, prvPAL_CreateFileForRx_CreateAnyFile) PASS
```

...

```
TEST(Full_OTA_PAL, prvPAL_CheckFileSignature_ValidSignature) PASS
TEST(Full_OTA_PAL, prvPAL_CheckFileSignature_InvalidSignatureBlockWritten) PASS
TEST(Full_OTA_PAL, prvPAL_CheckFileSignature_InvalidSignatureNoBlockWritten) PASS
TEST(Full_OTA_PAL, prvPAL_CheckFileSignature_NonexistingCodeSignerCertificate) PASS

-----

23 Tests 0 Failures 0 Ignored

OK

-----ALL TESTS FINISHED-----
```

OTA End-to-End Tests

To set up and run the end-to-end OTA tests

1. Follow the setup instructions in the README file ([<freertos>/tools/ota_e2e_test/README.md](#)).
2. Make sure that running the agent and PAL tests do not modify the `aws_clientcredential.h`, `aws_clientcredential_keys.h`, `aws_application_version.h`, or `aws_ota_codesigner_certificate.h` header files.
3. To run the OTA end-to-end test script, follow the example in the README file ([<freertos>/tools/ota_e2e_test/README.md](#)).

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the `device.json` file in the Device Tester configs folder.

After you have ported the FreeRTOS OTA library and the bootloader demo, you can start porting the Bluetooth Low Energy library. For instructions, see [Porting the Bluetooth Low Energy Library \(p. 76\)](#).

If your device does not support Bluetooth Low Energy functionality, then you are finished and can start the FreeRTOS qualification process. For more information, see the [FreeRTOS Qualification Guide](#).

Porting the Bluetooth Low Energy Library

You can use the FreeRTOS Bluetooth Low Energy library to provision Wi-Fi and send MQTT messages over Bluetooth Low Energy. The Bluetooth Low Energy library also includes higher-level APIs that you can use to communicate directly with the Bluetooth Low Energy stack. For more information, see [FreeRTOS Bluetooth Low Energy Library](#) in the FreeRTOS User Guide.

Note

A port of the FreeRTOS Bluetooth Low Energy library is currently not required for qualification.

Prerequisites

To port the Bluetooth Low Energy library, you need the following:

- An IDE project or CMakeLists.txt list file that includes the vendor-supplied Bluetooth Low Energy drivers.

For information about setting up a test project, see [Setting Up Your FreeRTOS Source Code for Porting](#) (p. 9).

- A validated configuration of the FreeRTOS kernel.

For information about configuring the FreeRTOS kernel for your platform, see [Configuring a FreeRTOS Kernel Port](#) (p. 28).

- A [Raspberry Pi 3 Model B+](#), with a memory card.

Porting

Three files in the `<freertos>/libraries/abstractions/ble_hal/include` folder define the FreeRTOS Bluetooth Low Energy APIs:

- `bt_hal_manager.h`
- `bt_hal_manager_adapter_ble.h`
- `bt_hal_gatt_server.h`

Each file includes comments that describe the APIs. You must implement the following APIs:

bt_hal_manager.h

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

bt_hal_manager_adapter_ble.h

- `pxRegisterBleApp`
- `pxUnregisterBleApp`

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

`bt_hal_gatt_server.h`

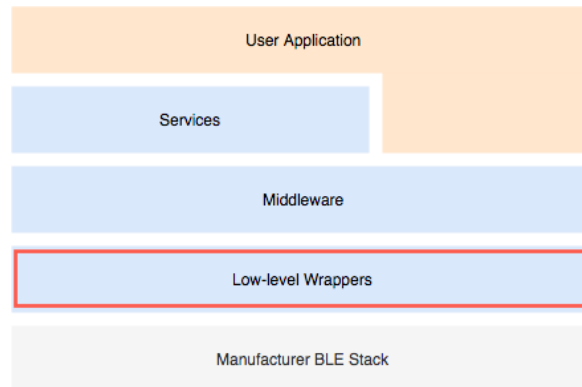
- `pxRegisterServer`
- `pxUnregisterServer`
- `pxGattServerInit`
- `pxAddService`
- `pxAddIncludedService`
- `pxAddCharacteristic`
- `pxSetVal`
- `pxAddDescriptor`
- `pxStartService`
- `pxStopService`
- `pxDeleteService`
- `pxSendIndication`
- `pxSendResponse`
- `pxMtuChangedCb`
- `pxCongestionCb`
- `pxIndicationSentCb`
- `pxRequestExecWriteCb`
- `pxRequestWriteCb`
- `pxRequestReadCb`
- `pxServiceDeletedCb`
- `pxServiceStoppedCb`
- `pxServiceStartedCb`
- `pxDescriptorAddedCb`
- `pxSetValCallbackCb`
- `pxCharacteristicAddedCb`
- `pxIncludedServiceAddedCb`
- `pxServiceAddedCb`
- `pxConnectionCb`
- `pxUnregisterServerCb`
- `pxRegisterServerCb`

Testing

This diagram shows the Bluetooth Low Energy testing framework.

To test your Bluetooth Low Energy ports, your computer communicates with an external, Bluetooth-enabled device (a Raspberry Pi 3 Model B+) over SSH, and with your device over Bluetooth Low Energy.

The Bluetooth Low Energy porting and qualification tests target the low-level wrapper layer that lies just above the manufacturer's hardware stack in the FreeRTOS Bluetooth Low Energy architecture:



If you are using an IDE to build test projects, you need to set up your library port in the IDE project.

Setting Up the IDE Test Project

If you are using an IDE for porting and testing, you need to add some source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their on-disk location. Do not create duplicate copies of source files.

To set up the Bluetooth Low Energy library in the IDE project

1. Add all of the files in `<freertos>/vendors/<vendor>/boards/<board>/ports/ble` to your `aws_tests` IDE project.
2. Add all of the files in `<freertos>/libraries/abstractions/ble_hal/include` to your `aws_tests` IDE project.
3. Add all of the files in `<freertos>/libraries/c_sdk/standard/ble` to your `aws_tests` IDE project.
4. Open `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/application_code/main.c`, and enable the required Bluetooth Low Energy drivers.

Configuring the CMakeLists.txt File

If you are using CMake to build your test project, you need to define a portable layer target for the library in your CMake list file.

To define a library's portable layer target in `CMakeLists.txt`, follow the instructions in [FreeRTOS Portable Layers \(p. 16\)](#).

The `CMakeLists.txt` template list file under `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes example portable layer target definitions. You can uncomment the definition for the library that you are porting, and modify it to fit your platform.

Setting Up Your Local Testing Environment

To set up the Raspberry Pi for testing

1. Follow the instructions in [Setting up your Raspberry Pi](#) to set up your Raspberry Pi with Raspbian OS.
2. Download bluez 5.50 from the [kernel.org repository](#).
3. Follow the instructions in the [README](#) on the kernel.org repository to install bluez 5.50 on the Raspberry Pi.
4. Enable SSH on the Raspberry Pi. For instructions, see the [Raspberry Pi documentation](#).
5. On your computer, open the `<freertos>/tests/bleTestsScripts/runPI.sh` script, and change the IP addresses in the first two lines to the IP address of your Raspberry Pi:

```
#!/bin/sh

scp * root@192.168.1.4:
ssh -t -t 192.168.1.4 -l root << 'ENDSSH'
rm -rf "/var/lib/bluetooth/*"
hciconfig hci0 reset
python test1.py
sleep 1
ENDSSH
```

Running the Tests

To execute the Bluetooth Low Energy tests

1. Execute the `runPI.sh` script.
2. Build the test project, and then flash it to your device for execution.
3. Check the test results in the UART console.

Validation

To officially qualify a device for FreeRTOS, you need to validate the device's ported source code with AWS IoT Device Tester. Follow the instructions in [Using AWS IoT Device Tester for FreeRTOS](#) in the FreeRTOS User Guide to set up Device Tester for port validation. To test a specific library's port, the correct test group must be enabled in the `device.json` file in the Device Tester `configs` folder.

After you have ported the Bluetooth Low Energy library, you can start the FreeRTOS qualification process. For more information, see the [FreeRTOS Qualification Guide](#).

Porting the Common I/O Libraries

In general, device drivers are independent of the underlying operating system and are specific to a given hardware configuration. A hardware abstraction layer (HAL) is a wrapper that provides common interfaces between drivers and higher-level application code. The HAL abstracts away the details of how a specific driver works and provides a uniform API to control similar devices. In this way, you can use the same APIs to control various devices across multiple microcontroller (MCU) based reference boards.

FreeRTOS common I/O acts as a hardware abstraction layer. It provides a set of standard APIs for accessing common serial devices across supported reference boards. These APIs communicate and

interact with some common peripherals and enable your application code to function across platforms. Without common I/O, the code that is required to work with low-level devices is silicon vendor specific.

Supported Peripherals

- UART
- SPI
- I2C

Supported Features

- Synchronous read/write – The function doesn't return until the requested amount of data has been transferred.
- Asynchronous read/write – The function returns immediately and the data transfer happens asynchronously. When function execution completes, a registered user callback is invoked.

Peripheral Specific

- I2C – Combine multiple operations into one transaction, typically to do write then read operations in one transaction.
- SPI – Transfer data between primary and secondary, which means the write and read operations happen simultaneously.

Porting

See the [FreeRTOS Porting Guide](#).

Topics

- [Prerequisites \(p. 81\)](#)
- [Testing \(p. 81\)](#)
- [Porting the I2C Library \(p. 83\)](#)
- [Porting the UART Library \(p. 86\)](#)
- [Porting the SPI Library \(p. 88\)](#)

Prerequisites

To port the common I/O Libraries, you need the following:

- An IDE project or CMakeLists.txt list file that includes the vendor-supplied I/O drivers.
- A validated configuration of the FreeRTOS kernel.

Testing

First, either set up an IDE project or configure CMake.

Set Up Your Local Testing Environment

No changes are required in the test file `<freertos>/libraries/abstractions/common_io/test/test_iot_peripheral.c`.

Device-specific code is in the following files

- `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h`
- `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c`

To set up your local testing environment

1. Create a test configuration header file named `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h`. For its content, refer to the "Test Setup" section for each peripheral.
2. Create a test setup file named `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c`. For its content, refer to the "Test Setup" section for each peripheral.
3. To enable the common I/O tests, open the file `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/aws_test_runner_config.h`.
4. Set `testrunnerFULL_COMMON_IO_ENABLED` to `1`

Set Up the IDE Test Project

If you're using an IDE for porting and testing, you must add the source files to the IDE test project before you can test your ported code.

Important

In the following steps, make sure that you add the source files to your IDE project from their current on-disk location. Don't create duplicate copies of the source files.

To set up the common I/O libraries in IDE project

1. Add all implementation source files `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_<peripheral>.c` to your `aws_tests` IDE project (one file per peripheral).
2. Add all test source files `<freertos>/libraries/abstractions/common_io/test/test_iot_<peripheral>.c` to your `aws_tests` IDE project (one file per peripheral).
3. Add the test configuration file `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h` to your `aws_tests` IDE project (just this one file for all peripherals).
4. Add the test setup file `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c` to your `aws_tests` IDE project (just this one file for all peripherals).

Configure the CMakeLists.txt File

If you're using CMake to build your test project, you must define a portable layer target for the library in your CMake list file.

The `CMakeLists.txt` template list file at `<freertos>/vendors/<vendor>/boards/<board>/CMakeLists.txt` includes examples of portable layer target definitions. Uncomment the definition of each library that you're porting, and modify it to fit your platform.

Example

The following is portable layer target definition for the common I/O library.

```
# Common I/O
```

```
afr_mcu_port(common_io)
target_sources(
    AFR::common_io::mcu_port
    INTERFACE
        # peripheral implementations
        <freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_<peripheral_1>.c
        <freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_<peripheral_2>.c
        <freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_<peripheral_3>.c
        ...
        # test configuration and pre-steps
        <freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c
    )
#
-----
# FreeRTOS demos and tests
#
-----
...

if(AFR_IS_TESTING)
    set(exe_target aws_tests)
else()
    set(exe_target aws_demos)
endif()
...

# link common io library along with others
target_link_libraries(
    ${exe_target}
    PRIVATE
        AFR::wifi
        AFR::utils
        AFR::common_io
)
)
```

Run the Tests

To execute the common I/O tests

1. Build the test project, and then flash it to your device for execution.
2. Check the test results in the UART console.

Porting the I2C Library

I2C library interfaces with vendor-supplied I2C drivers. If the device doesn't have an I2C peripheral, you can skip porting I2C interfaces. The I2C library can only use the I2C peripheral that is on the device as the primary.

Prerequisites

To port the I2C library, you need an I2C secondary device. It can be one of the following:

- An onboard I2C sensor.
- An external device, such as a Raspberry PI.

Porting

Use the vendor-supplied I2C driver library to implement all the functions in `<freertos>/libraries/abstractions/common_io/include/iot_i2c.h`. The header file provides the required API behavior information. An implementation source file should be created and named `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_i2c.c`.

If any of the I2C features are not supported on a target device, make the corresponding functions return `IOT_I2C_FUNCTION_NOT_SUPPORTED`. For the list of functions that can return `IOT_I2C_FUNCTION_NOT_SUPPORTED`, see the API as documented in `<freertos>/libraries/abstractions/common_io/include/iot_i2c.h`.

Anonymous Handle "struct IotI2CDescriptor"

This usually encapsulates the driver's handle and variety of states. See the following example.

```
/* Suppose the data type of driver handle for I2C is Driver_I2C_Handle */
struct IotI2CDescriptor
{
    Driver_I2C_Handle xHandle;        /* Driver Handle. */
    IotI2CConfig_t xConfig;           /* Bus Configuration. */
    IotI2CCallback_t xCallback;       /* Callback function. */
    void * pvUserContext;             /* User context passed in callback. */
    uint16_t usSlaveAddr;             /* Slave Address. */
    uint16_t usTransmittedTxBytes;    /* Number of Transmitted Bytes */
    uint16_t usReceivedRxBytes;       /* Number of Received Bytes */
    SemaphoreHandle_t xSemphr;        /* Optional, useful when there is a synchronization
    situation. */
    /* State: if already opened. */
    /* State: if send no stop. */
};
```

Test Setup

Hardware Setup

If you're using an onboard sensor as a slave device, you can skip this step.

If you use an external device, you need to wire the SDA (data) lines and SCL (clock) lines of the two devices.

You can find the I2C test file in the following directory: `<freertos>/libraries/abstractions/common_io/test/test_iot_i2c.c`

To test the set up configurations

1. Add the I2C configurations to `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h`.

`IOT_TEST_COMMON_IO_I2C_SUPPORTED`

If this device has I2C peripheral, set to **1**. Otherwise, set it to **0**.

`IOT_TEST_COMMON_IO_I2C_SUPPORTED_SEND_NO_STOP`

If the I2C doesn't explicitly support sending stop condition, set to **1**. Otherwise, set to **0**.

`IOT_TEST_COMMON_IO_I2C_SUPPORTED_CANCEL`

If the I2C supports cancelling the asynchronous transaction with interrupt or DMA, set to **1**. Otherwise, set to **0**.

`I2C_TEST_SET`

Specify the number of I2C instances to test.

2. Define test data in the `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h` file.

`i2cTestInstanceId`

The I2C instance IDs.

`i2cTestInstanceNum`

The total number of I2C instances.

`i2cTestSlaveAddr`

Device address.

`i2cTestDeviceRegister`

Register address on the test device.

`i2cTestWriteVal`

A byte value to be written to the test device.

`gIotI2CHandle`

Not used. Define it as an array of null to compile.

Example

```
/* I2C includes */
#include "iot_i2c.h"

#define IOT_TEST_COMMON_IO_I2C_SUPPORTED 1

#if ( IOT_TEST_COMMON_IO_I2C_SUPPORTED == 1 )
    #define IOT_TEST_COMMON_IO_I2C_SUPPORTED_SEND_NO_STOP 1
    #define IOT_TEST_COMMON_IO_I2C_SUPPORTED_CANCEL 1
#endif

#define I2C_TEST_SET 1

/* Slave address. */
const uint8_t i2cTestSlaveAddr[ I2C_TEST_SET ] = { 0xD4 };
/* Register address. */
const uint8_t i2cTestDeviceRegister[ I2C_TEST_SET ] = { 0x73 };
/* A value that is written to slave device during test. */
const uint8_t i2cTestWriteVal[ I2C_TEST_SET ] = { 0b01101010 };
/* I2C instance ID. */
const uint8_t i2cTestInstanceId[ I2C_TEST_SET ] = { 1 };
/* Total number of I2C instances. */
const uint8_t i2cTestInstanceNum[ I2C_TEST_SET ] = { 3 };

/* Unused, but this needs to be defined. */
IotI2CHandle_t gIotI2CHandle[ 4 ] = { NULL, NULL, NULL, NULL };
```

3. Add I2C test setup code to the `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c` file.

```
#include "test_iot_internal.h"

/* These global variables are defined in test_iot_i2c.c. */
extern uint8_t uctestIotI2CSlaveAddr;
extern uint8_t xtestIotI2CDeviceRegister;
extern uint8_t uctestIotI2CWriteVal;
extern uint8_t uctestIotI2CInstanceId;
```

```
extern uint8_t uctestIotI2CInstanceNum;

void SET_TEST_IOT_I2C_CONFIG(int testSet)
{
    uctestIotI2CSlaveAddr = i2cTestSlaveAddr[testSet];
    xtestIotI2CDeviceRegister = i2cTestDeviceRegister[testSet];
    uctestIotI2CWriteVal = i2cTestWriteVal[testSet];
    uctestIotI2CInstanceIdx = i2cTestInstanceIdx[testSet];
    uctestIotI2CInstanceNum = i2cTestInstanceNum[testSet];
}
```

Porting the UART Library

The UART library interfaces with vendor-supplied UART drivers. If the device doesn't have any UART peripherals, you can skip porting the UART interface.

Prerequisites

- Use a jump wire to connect the RX and TX of the UART for loopback testing.

Porting

Use the vendor-supplied UART driver library to implement all the functions in `<freertos>/libraries/abstractions/common_io/include/iot_uart.h`. The header file provides information about the required API behavior. An implementation source file should be created and named `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_uart.c`.

If the target device doesn't support any UART features, make the corresponding functions return `IOT_UART_FUNCTION_NOT_SUPPORTED`. For the list of functions that can return `IOT_UART_FUNCTION_NOT_SUPPORTED`, refer to the API as documented in `<freertos>/libraries/abstractions/common_io/include/iot_uart.h`.

Anonymous Handle "struct IotUARTDescriptor"

This usually encapsulates driver's handle and variety of states. See the following example.

```
/* Suppose the data type of the driver handle for UART is UART_Handle */
struct IotUARTDescriptor
{
    IotUARTCallback_t xUartCallback; /* Application Specified callback. */
    UART_Handle * pxUartContext; /* UART handle to be passed to driver functions. */
    void * pvUserCallbackContext; /
    uint8_t sOpened;
};
```

Test Setup

Hardware Setup

On the UART port to test, connect the TX and RX for loopback by using a jump wire.

You can find the UART test file in the following directory: `<freertos>/libraries/abstractions/common_io/test/test_iot_uart.c`

To test the setup configurations

1. Add the UART configuration to the `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h` file.

IOT_TEST_COMMON_IO_UART_SUPPORTED

If this device has UART peripheral, set to 1. Otherwise, set 0.

UART_TEST_SET

The number of UART instances to test.

2. Define test data in the `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h` file.

uartTestPort

The UART instance IDs.

uartIotUartFlowControl

The UART flow control configuration.

uartIotUartParity

The UART parity bit configuration.

uartIotUartWordLength

The UART word length configuration.

uartIotUartStopBits

The UART stop bit configuration.

Example

```
/* UART */
#define UART_TEST_SET 1
const uint8_t uartTestPort[ UART_TEST_SET ] = { 1 };
const uint32_t uartIotUartFlowControl[ UART_TEST_SET ] = { UART_FLOW_CONTROL };
const uint32_t uartIotUartParity[ UART_TEST_SET ] = { UART_PARITY };
const uint32_t uartIotUartWordLength[ UART_TEST_SET ] = { UART_WORD_LENGTH };
const uint32_t uartIotUartStopBits[ UART_TEST_SET ] = { UART_STOP_BITS };
```

3. Add the UART test setup code to the `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c` file.

```
#include "test_iot_internal.h"

/* UART */
extern uint8_t uctestIotUartPort;
extern uint32_t ulctestIotUartFlowControl;
extern uint32_t ulctestIotUartParity;
extern uint32_t ulctestIotUartWordLength;
extern uint32_t ulctestIotUartStopBits;

void SET_TEST_IOT_UART_CONFIG( int testSet )
{
    uctestIotUartPort = uartTestPort[ testSet ];
    ulctestIotUartFlowControl = uartIotUartFlowControl[ testSet ];
    ulctestIotUartParity = uartIotUartParity[ testSet ];
    ulctestIotUartWordLength = uartIotUartWordLength[ testSet ];
    ulctestIotUartStopBits = uartIotUartStopBits[ testSet ];
}
```

Porting the SPI Library

The SPI library interfaces with vendor-supplied SPI drivers. If the device doesn't have an SPI peripheral, you can skip porting the SPI interfaces. The SPI library can only use the SPI peripheral on the device as master.

Porting

Use the vendor-supplied SPI driver library to implement all the functions in `<freertos>/libraries/abstractions/common_io/include/iot_spi.h`. The header file provides the required API behavior information. The implementation source file should be created as `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/iot_spi.c`.

It's possible that a target device doesn't support some SPI features. In that case, make the corresponding functions return `IOT_SPI_FUNCTION_NOT_SUPPORTED`. For the list of functions that can return `IOT_SPI_FUNCTION_NOT_SUPPORTED`, refer to the API as documented in `<freertos>/libraries/abstractions/common_io/include/iot_spi.h`.

Anonymous Handle "struct IotSPIDescriptor"

This usually encapsulates the driver's handle and variety of states. See the following example.

```
/* Suppose the data type of driver handle for SPI is Driver_SPI_Handle */
struct IotSPIDescriptor
{
    Driver_SPI_Handle xHandle;    /* Driver Handle. */
    IotSPIConfig_t xConfig;      /* Bus Configuration. */
    IotSPICallback_t xCallback;  /* Callback function. */
    void * pvUserContext;        /* User context passed in callback. */
    /* State: if already opened. */
};
```

Test Setup

You can find the SPI test file in the following directory: `<freertos>/libraries/abstractions/common_io/test/test_iot_spi.c`

To test the setup configurations

1. Add the SPI configurations to the `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h` file.

`IOT_TEST_COMMON_IO_SPI_SUPPORTED`

If the device has an SPI peripheral, set this to **1**. Otherwise, set to **0**.

`SPI_TEST_SET`

The number of SPI instances to test.

2. Define test data in the `<freertos>/vendors/<vendor>/boards/<board>/aws_tests/config_files/test_iot_config.h` file.

`spiTestPort`

The SPI instance IDs.

`spiIotMode`

The SPI mode.

`spiIotSpitBitOrder`

The SPI bit order.

`spiIotFrequency`

The SPI frequency.

`spiIotDummyValue`

The dummy value.

Example

```
/* SPI includes */
#include "iot_spi.h"

#define IOT_TEST_COMMON_IO_SPI_SUPPORTED 1
#define I2C_TEST_SET 1

const uint8_t spiTestPort[ SPI_TEST_SET ] = { 1 };
const uint32_t spiIotMode[ SPI_TEST_SET ] = { eSPIMode0 };
const uint32_t spiIotSpitBitOrder[ SPI_TEST_SET ] = { eSPIMSBFirst };
const uint32_t spiIotFrequency[ SPI_TEST_SET ] = { 500000U };
const uint32_t spiIotDummyValue[ SPI_TEST_SET ] = { 0 };
```

3. Add SPI test setup code in the `<freertos>/vendors/<vendor>/boards/<board>/ports/common_io/test_iot_internal.c` file.

```
#include "test_iot_internal.h"

/* SPI */
extern uint8_t ultestIotSpiInstance;
extern IotSPIMode_t xtestIotSPIDefaultConfigMode;
extern IotSPIBitOrder_t xtestIotSPIDefaultconfigBitOrder;
extern uint32_t ultestIotSPIFrequency;
extern uint32_t ultestIotSPIDummyValue;

void SET_TEST_IOT_SPI_CONFIG(int testSet)
{
    ultestIotSpiInstance = spiTestPort[ testSet ];
    xtestIotSPIDefaultConfigMode = spiIotMode[ testSet ];
    xtestIotSPIDefaultconfigBitOrder = spiIotSpitBitOrder[ testSet ];
    ultestIotSPIFrequency = spiIotFrequency[ testSet ];
    ultestIotSPIDummyValue = spiIotDummyValue[ testSet ];
}
```

Migrating from Version 1.4.x to Version 201906.00 (and newer)

For a list of FreeRTOS releases, see <https://github.com/aws/amazon-freertos/releases>.

Migrating Applications

FreeRTOS version 201906.00 introduced some changes to the FreeRTOS directory structure that break project files built on previous versions of FreeRTOS. In order for applications built on previous versions of FreeRTOS to work with FreeRTOS version 201906.00 or newer, you must move the application code to new projects and include the 201906.00 header files in the application.

Version 201906.00 introduced new APIs for the MQTT, Device Shadow, and Device Defender libraries. The APIs for previous versions of these libraries are accessible through header files of the 201906.00 implementations of these libraries, making FreeRTOS version 201906.00 backward-compatible.

Note

If you are migrating from previous versions to version 201906.00 or newer, you might need to reconfigure your `iot_config.h` and `FreeRTOSConfig.h` files to accommodate the new library implementations. For information about global configuration settings, see the [Global Configuration File Reference](#).

Migrating Ports

If you have ported a version of FreeRTOS released prior to the 201906.00 release, you need to migrate your ported code to be compatible with versions 201906.00 and later. For information about porting, see the [FreeRTOS Porting Guide](#).

FreeRTOS Code Directory Structure

In versions released prior to 201906.00, the `<freertos>/lib/third_party/mcu_vendor/vendor` folder held the vendor-ported code. One or more project files under the same `vendor` folder compiled the code. In versions 201906.00 and later, vendor code is located under the `<freertos>/vendors/vendor` folder, and project files are located under the `<freertos>/projects/vendor` folder.

Note

The code for ports did not change with version 201906.00. Only the location of the code changed. Move any existing ports to the new folder structure.

CMake Build System

Version 201906.00 introduced support for using CMake to generate project files. For information about using CMake with FreeRTOS, see [Building FreeRTOS with CMake \(p. 20\)](#).

A CMake list file is required for qualification. For information about creating a CMake list file, see [Creating a CMake List File \(p. 12\)](#).

Migrating the Wi-Fi Library Port

The FreeRTOS Wi-Fi library features four new APIs to add, remove, and retrieve a Wi-Fi network, and to receive notifications for Wi-Fi network state changes. All of these new APIs are optional, and are intended to support Wi-Fi credentials provisioning over Bluetooth Low Energy. If your device does not support Bluetooth Low Energy, you do not need to implement these APIs.

- `WIFI_NetworkAdd`

```
WIFIReturnCode_t WIFI_NetworkAdd(  
    const WIFINetworkProfile_t * const pxNetworkProfile,  
    uint16_t * pusIndex );
```

- `WIFI_NetworkGet`

```
WIFIReturnCode_t WIFI_NetworkGet(  
    WIFINetworkProfile_t * pxNetworkProfile,  
    uint16_t usIndex );
```

- `WIFI_NetworkDelete`

```
WIFIReturnCode_t WIFI_NetworkDelete( uint16_t usIndex );
```

- `WIFI_RegisterNetworkStateChangeEventCallback`

```
WIFIReturnCode_t WIFI_RegisterNetworkStateChangeEventCallback(  
    IotNetworkStateChangeEventCallback_t xCallback );
```

With the following typedef statements:

```
typedef void ( *IotNetworkStateChangeEventCallback_t ) (  
    uint32_t ulNetworkType,  
    AwsIotNetworkState_t xState );
```

```
typedef enum AwsIotNetworkState  
{  
    eNetworkStateUnknown = 0,  
    eNetworkStateDisabled,  
    eNetworkStateEnabled  
} AwsIotNetworkState_t;
```

For information about porting the Wi-Fi library, see [Porting the Wi-Fi Library](#) in the FreeRTOS Porting Guide.