Amazon FreeRTOS

Porting Guide

Amazon FreeRTOS: Porting Guide

Table of Contents

Porting Amazon FreeRTOS to Your Device

Before a microcontroller board can run Amazon FreeRTOS, some Amazon FreeRTOS code must be ported to the device's hardware.

To port Amazon FreeRTOS to your device

1. Follow the instructions in Downloading Amazon FreeRTOS for Porting (p. 6) to download the latest version of Amazon FreeRTOS for porting.
2. Follow the instructions in Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7) to configure the files and folders in your Amazon FreeRTOS download for porting and testing.
3. Follow the instructions in Porting the Amazon FreeRTOS Libraries (p. 13) to port the Amazon FreeRTOS libraries to your device. Each porting topic includes instructions on testing the ports.

System Requirements

The device that you port to Amazon 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. 53)) Two executable images stored on the MCU

Porting Older Versions of Amazon FreeRTOS

If you are porting an older version of Amazon FreeRTOS, go to the `amazon-freertos` GitHub repository, and checkout the version of Amazon 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 Amazon FreeRTOS Porting and Qualification Documentation

Date	Porting and Qualification Documentation Version	Change History	Amazon FreeRTOS Version
May 21, 2019	1.4.8 (Porting Guide) 1.4.8 (Qualification Guide)	• Porting documentation moved to the Amazon FreeRTOS Porting Guide • Qualification documentation moved to the Amazon FreeRTOS Qualification Guide	1.4.8

Date	Porting and Qualification Documentation Version	Change History	Amazon FreeRTOS Version
February 25, 2019	1.1.6	• 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	• Updated Checklist for Qualification appendix with CMake requirement (page 70)	1.4.5 1.4.6
December 12, 2018	1.1.4	• Added lwIP porting instructions to TCP/IP porting appendix (page 31)	1.4.5
November 26, 2018	1.1.3	• Added BLE porting appendix (page 52) • Added AWS IoT Device Tester for Amazon FreeRTOS testing information throughout document • Added CMake link to Information for listing on the Amazon FreeRTOS Console appendix (page 85)	1.4.4
November 7, 2018	1.1.2	• Updated PKCS #11 PAL interface porting instructions in PKCS #11 porting appendix (page 38) • Updated path to `CertificateConfigurator.html` (page 76) • Updated Getting Started Guide Template appendix (page 80)	1.4.3

Date	Porting and Qualification Documentation Version	Change History	Amazon FreeRTOS Version
October 8, 2018	1.1.1	• Added new "Required for AFQP" column to `aws_test_runner_config.h` 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	• 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	Amazon FreeRTOS Version
August 9, 2018	1.0.1	• 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 Amazon FreeRTOS Qualification Program Guide	1.3.0

Porting FAQs

What is an Amazon FreeRTOS port?

An Amazon FreeRTOS port is a board-specific implementation of APIs for the required Amazon FreeRTOS libraries and the Amazon 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 (BLE), or over-the-air (OTA) updates. Are all libraries required to port Amazon FreeRTOS?

The primary requirement for Amazon FreeRTOS is that your device can connect to the AWS Cloud. If, for example, you can connect to the cloud across a secure ethernet connection, Amazon 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 Amazon FreeRTOS port tests?

The following network connections are required to run the Amazon 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 Amazon FreeRTOS Porting Guide, please contact the Amazon FreeRTOS engineering team.

Downloading Amazon FreeRTOS for Porting

Before you begin porting Amazon FreeRTOS to your platform, you need to download Amazon FreeRTOS or clone the Amazon FreeRTOS repository from GitHub.

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.

To download Amazon FreeRTOS from GitHub

1. Navigate to the amazon-freertos GitHub repository.
2. Click on **Branch: master**, switch the checkout mode from **Branches** to **Tags**, and choose the newest release tag.
3. With the latest version checked out, click on **Clone or download**, and then choose **Download ZIP** to download the Amazon FreeRTOS version as a ZIP file.

To clone the Amazon FreeRTOS repository

1. If you have not already done so, install git on your machine.

   Note
   In these instructions, we use the `git` command-line interface. You can also use a git client of your choice.
2. Navigate to the amazon-freertos GitHub repository.
3. Click on **Clone or download**, and then copy the URL listed beneath **Clone with HTTPS**.
4. Open a terminal or command line window, and change directories to a location where you want to keep your local copy of the Amazon FreeRTOS repository's files.
5. Issue the following command to clone the repository to your current directory:

```
git clone https://github.com/aws/amazon-freertos.git
```

6. Change directories to the `amazon-freertos` folder, and checkout the latest version:

```
cd amazon-freertos
```

```
git checkout version
```

After you download or clone Amazon FreeRTOS, you can start porting Amazon FreeRTOS code to your platform. For instructions, see Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7), and then see Porting the Amazon FreeRTOS Libraries (p. 13).

Note
Throughout Amazon FreeRTOS documentation, the Amazon FreeRTOS download is referred to as *<amazon-freertos>*.

Setting Up Your Amazon FreeRTOS Source Code for Porting

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

To prepare your Amazon FreeRTOS download for porting, you need to follow the instructions in Configuring the Amazon FreeRTOS Download (p. 7) to configure the directory structure of your Amazon 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 Amazon FreeRTOS Source Code for Testing (p. 9).

> **Note**
> You must use the AWS IoT Device Tester for Amazon FreeRTOS to officially validate your ports for qualification. For more information about AWS IoT Device Tester for Amazon FreeRTOS, see Using AWS IoT Device Tester for Amazon FreeRTOS in the Amazon FreeRTOS User Guide.
> For information about qualifying your device for Amazon FreeRTOS, see the Amazon FreeRTOS Qualification Guide.

After you configure your Amazon FreeRTOS download and set up your testing environment, you can beginning porting Amazon FreeRTOS. For porting and testing instructions, see Porting the Amazon FreeRTOS Libraries (p. 13).

Configuring the Amazon FreeRTOS Download

All qualified Amazon FreeRTOS ports use the same directory structure. New files must be created in the correct folder locations.

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

Configuring Directories for Vendor-supplied, Board-specific Libraries

Under the download's root directory (*<amazon-freertos>*), there are five folders: `cmake`, `demos`, `lib`, `tests`, and `tools`.

```
<amazon-freertos>
+ - cmake    (Contains files for CMake support)
+ - demos    (Contains projects that build demo applications)
+ - lib    (Contains Amazon FreeRTOS and third-party libraries)
+ - tests    (Contains projects that build porting tests)
+ - tools    (Contains tools for configuration, development, and testing)
```

The `tests` folder is structured as follows:

```
<amazon-freertos>
+ - tests
    + - common    (Contains files built by all test projects)
    + - pc    (Contains a reference test project for the FreeRTOS Windows port)
    + - <vendor>    (Template, to be renamed to the name of the MCU vendor)
```

```
        + - <board>    (Template, to be renamed to the name of the development board)
```

The *<board>* folder is a template folder provided to simplify the creation of a new test project. This template folder's directory structure ensures all test projects have a consistent organization. The *<board>* folder has the following structure:

```
<amazon-freertos>
+ - tests
    + - <vendor>
        + - <board>
            + - common
            |   + - application_code    (Contains main.c, which contains main())
            |   |   + - <vendor>_code    (Contains vendor-supplied, board-specific files)
            |   + - config_files    (Contains Amazon FreeRTOS config files)
            + - <ide>    (Contains an IDE-specific project)
```

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.

To set up the directories for vendor-supplied, board-specific libraries

1. Save all required vendor-supplied libraries that are specific to the board in the *<vendor>*_code folder.
2. Rename the *<vendor>* in the *<vendor>*_code folder to the name of the vendor.
3. Rename the *<ide>* folder to the name of the IDE that you are using to build the test project.
4. Rename the *<vendor>* folder to the name of the vendor, and rename the *<board>* folder to the name of the development board.

Configuring Directories for Common Vendor-supplied Libraries

Other vendor-supplied files, such as a GPIO library, are common across a board's MCU family. These files belong in the *<amazon-freertos>*/lib/third_party/mcu_vendor folder, which has the following structure:

```
<amazon-freertos>
+ - lib
    + - third_party    (Contains all non-board-specific, third-party libraries)
        + - mcu_vendor    (Contains vendor-supplied, MCU-specific libraries)
            + - <vendor>    (Template, to be renamed to the name of the MCU vendor)
                + - <driver_library>    (Template, to be renamed to the library name)
                    + - <driver_library_version>    (Template, to be renamed to the library
 version)
```

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 *<amazon-freertos>*/tests or *<amazon-freertos>*/demos.

Configuring FreeRTOSConfig.h

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

To configure your board name in FreeRTOSConfig.h

1. Open *<amazon-freertos>*/demos/*<vendor>*/*<board>*/common/config_files/FreeRTOSConfig.h.
2. In the line #define configPLATFORM_NAME "*<Unknown>*", change *<Unknown>* to match the name of your board.

Setting Up Your Amazon FreeRTOS Source Code for Testing

Amazon FreeRTOS includes tests for each ported library in folders under *<amazon-freertos>*/tests/common. The tests/common/test_runner/aws_test_runner.c defines a RunTests function that runs each test that you have specified in the /tests/vendor/board/common/config_files/aws_test_runner_config.h header file. As you port each Amazon FreeRTOS library, you can test the ports by building the ported Amazon FreeRTOS source code, flashing the compiled code to your board, and running it on the board.

To build the Amazon FreeRTOS source code for testing, use a supported IDE. For instructions on creating an IDE project for development and testing, see Creating an IDE Project (p. 9).

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 you do not need to flash your code manually if you are using Device Tester.

Creating an IDE Project

After you configure your Amazon 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. These instructions were written using an Eclipse-based IDE, but Amazon FreeRTOS test projects look the same in most IDEs.

> **Important**
> You must import all files in their original position in the directory structure. Never directly copy files into the project's folder, and never use absolute file paths.
> 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 *<amazon-freertos>*/tests/*<vendor>*/*<board>*/*<ide>* directory.
2. In the IDE, create three virtual folders under the aws_tests project:
   - application_code
   - config_files
   - lib

Under `aws_tests`, there should now be three virtual folders in the IDE project:

```
aws_tests    (The project name)
+ - application_code    (Contains application logic, in this case, porting test code)
+ - config_files    (Contains header files that configure Amazon FreeRTOS libraries)
+ - lib    (Contains Amazon and third-party libraries)
```

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

3. Under `aws_tests/application_code`, create a virtual folder named `common_test`, and import the following directories into that virtual folder:

   - *<amazon-freertos>*`/tests/common/framework`
   - *<amazon-freertos>*`/tests/common/include`
   - *<amazon-freertos>*`/tests/common/memory_leak`
   - *<amazon-freertos>*`/tests/common/test_runner`
   - *<amazon-freertos>*`/tests/common/utils`

4. Import the *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/application_code/`*<vendor>*`_code` directory and the *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/application_code/main.c` file directly into the `aws_tests/application_code` virtual folder.

   The `application_code` virtual folder should now look like this in the IDE project:

```
aws_tests
+ - application_code
    + - common_test
    |   + - framework
    |   + - include
    |   + - memory_leak
    |   + - test_runner
    |   + - utils
    + - <vendor>_code
    + - main.c
```

5. Import all of the header files in the *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/config_files` directory into the `aws_tests/config_files` virtual folder.

   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.

   The `config_files` virtual folder should now look something like this in the IDE project:

```
aws_tests
+ - config_files
    + - aws_bufferpool_config.h
    + - aws_ggd_config.h
    + - aws_mqtt_agent_config.h
    + - aws_mqtt_config.h
    + - aws_ota_agent_config.h
    + - aws_pkcs11_config.h
    + - aws_secure_sockets_config.h
```

```
    + - aws_shadow_config.h
    + - aws_test_ota_config.h
    + - aws_test_pkcs11_config.h
    + - aws_test_runner_config.h
    + - aws_test_tcp_config.h
    + - aws_test_wifi_config.h
    + - aws_wifi_config.h
    + - FreeRTOSConfig.h
    + - FreeRTOSIPConfig.h
    + - trcConfig.h
    + - trcSnapshotConfig.h
    + - unity_config.h
```

6. Under `aws_tests/lib`, create a virtual folder named `aws`. As you port Amazon FreeRTOS, you create virtual folders under this directory for each library.
7. Import the directory *<amazon-freertos>*`/lib/include` into the `aws_tests/lib/aws` virtual folder.
8. Under `aws_tests/lib/aws`, create virtual folders named `FreeRTOS` and `Utils`.
9. Import *<amazon-freertos>*`/lib/utils/aws_system_init.c` into the `aws_tests/lib/aws/Utils` virtual folder.
10. Import the FreeRTOS memory management implementation that you are using for your device into the `aws_tests/lib/aws/FreeRTOS` virtual folder. The *<amazon-freertos>*`/lib/FreeRTOS/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.
11. Import all of the files in the *<amazon-freertos>*`/lib/FreeRTOS` directory into the `aws_tests/lib/aws/FreeRTOS` virtual folder.
12. Under `aws_tests/lib/aws/FreeRTOS`, create a virtual folder named `portable`, and then import the subdirectory of *<amazon-freertos>*`/lib/FreeRTOS/portable` that corresponds to your compiler and architecture into the `portable` virtual folder.

    The `lib/aws` virtual folder should now look like this in the IDE project:

```
aws_tests
+ - lib
    + - aws
        + - FreeRTOS
        |   + - portable
        |   |   + - MemMang
        |   |   |   + - heap_4.c
        |   |   + - MSVC-MingW
        |   |       + - port.c
        |   |       + - portmacro.h
        |   + - event_groups.c
        |   + - list.c
        |   + - queue.c
        |   + - tasks.c
        |   + - timers.c
        + - include
        + - Utils
        |   + - aws_system_init.c
        + - unity_config.h
```

13. Under `aws_tests/lib`, create a virtual folder named `third_party`, and then import all of the files in *<amazon-freertos>*`/lib/third_party/`*<mcu_vendor>*`/`*<vendor>*`/`*<driver_library>*`/`*<driver_library_version>* into the `third_party` virtual folder.
14. Under `aws_tests/lib/third_party`, create virtual folders named `unity` and `unity_fixture`.

15. Import all of the files in *<amazon-freertos>*/lib/third_party/unity/src into the aws_tests/lib/third_party/unity virtual folder.
16. Import all of the files in *<amazon-freertos>*/lib/third_party/unity/extras/fixture/src into the aws_tests/lib/third_party/unity_fixture virtual folder.

    The lib/third_party virtual folder should now look like this in the IDE project:

```
aws_tests
+ - lib
    + - third_party
        + - unity
        |   + - unity_internals.h
        |   + - unity.c
        |   + - unity.h
        + - unity_fixture
            + - unity_fixture_internals.h
            + - unity_fixture.c
            + - unity_fixture.h
            + - unity_fixture_malloc_overrides.h
```

17. Open your project's IDE properties, and add the following paths to your compiler's include path:

    - *<amazon-freertos>*/tests/common/include
    - *<amazon-freertos>*/lib/include
    - *<amazon-freertos>*/lib/include/private
    - *<amazon-freertos>*/lib/FreeRTOS/portable/*<compiler>*/*<architecture>*
    - *<amazon-freertos>*/lib/third_party/unity/src
    - *<amazon-freertos>*/lib/third_party/unity/extras/fixture/src
    - *<amazon-freertos>*/demos/*<vendor>*/*<board>*/common/config_files
    - Any paths required for vendor-supplied driver libraries
18. 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 Amazon FreeRTOS libraries to your device. For instructions, see Porting the Amazon FreeRTOS Libraries (p. 13).

Porting the Amazon FreeRTOS Libraries

Before you start porting, follow the instructions in Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7).

The following diagram visualizes the porting process.

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

1. Implementing the `configPRINT_STRING()` macro (p. 13)
2. Configuring a FreeRTOS Kernel Port (p. 14)
3. Porting the Wi-Fi Library (p. 16)

   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 Amazon FreeRTOS Wi-Fi library is not necessarily required.
4. Porting a TCP/IP Stack (p. 20)
5. Porting the Secure Sockets Library (p. 24)
6. Porting the PKCS #11 Library (p. 29)
7. Porting the TLS Library (p. 33)
8. Setting Up the MQTT Library for Testing (p. 51)
9. Porting the OTA Library (p. 53)

   Note
   A port of the Amazon FreeRTOS OTA update library is currently not required for qualification.
10. Porting the BLE Library (p. 57)

    Note
    A port of the Amazon FreeRTOS BLE library is currently not required for qualification.

After you port Amazon FreeRTOS to your board, you can officially validate the ports for Amazon FreeRTOS qualification with AWS IoT Device Tester for Amazon FreeRTOS. For more information about AWS IoT Device Tester for Amazon FreeRTOS, see Using AWS IoT Device Tester for Amazon FreeRTOS in the Amazon FreeRTOS User Guide. For information about qualifying your device for Amazon FreeRTOS, see the Amazon FreeRTOS Qualification Guide.

Amazon FreeRTOS Porting Flowchart

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

Implementing the `configPRINT_STRING()` macro

You must implement the `configPRINT_STRING()` macro before you port the Amazon FreeRTOS libraries. Amazon 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.
- An Amazon FreeRTOS project configured for your platform, and a porting-test IDE project.

  For information, see Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7).

Implementation

To implement `configPRINT_STRING()`

1. Connect your device to a terminal emulator to output test results.
2. Open the file *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/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 *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/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. 14) for instructions.

Configuring a FreeRTOS Kernel Port

This section provides instructions for integrating a port of the FreeRTOS kernel into an Amazon FreeRTOS port-testing project. For a list of available kernel ports, see Official FreeRTOS Ports.

Amazon FreeRTOS uses the FreeRTOS kernel for multitasking and inter-task communications. For more information, see the FreeRTOS Kernel Fundamentals in the Amazon 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 Amazon FreeRTOS engineering team.

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 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 Amazon FreeRTOS Source Code for Porting (p. 7).
- An implementation of the `configPRINT_STRING()` macro for your device.

  For information about implementing `configPRINT_STRING()`, see Implementing the `configPRINT_STRING()` macro (p. 13).

Configuring the FreeRTOS Kernel

The header file *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/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
`configCPU_CLOCK_HZ`	Specifies the frequency of the clock used to generate the tick interrupt.
`configMINIMAL_STACK_SIZE`	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 `configMINIMAL_STACK_SIZE` if overflows occur. To save RAM, set stack sizes to the minimum value that does not result in a stack overflow.
`configTOTAL_HEAP_SIZE`	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 *<amazon-freertos>*/lib/utils/aws_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 *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/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. 16) for instructions.

Porting the Wi-Fi Library

The Amazon FreeRTOS Wi-Fi library interfaces with vendor-supplied Wi-Fi drivers. For more information about the Amazon FreeRTOS Wi-Fi library, see Amazon FreeRTOS Wi-Fi Library in the Amazon FreeRTOS User Guide.

If your device does not support Wi-Fi networking, you can skip porting the Amazon FreeRTOS Wi-Fi library and start Porting a TCP/IP Stack (p. 20).

> **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 Amazon FreeRTOS Wi-Fi library is not necessarily required.

Prerequisites

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

- An IDE project that includes the vendor-supplied Wi-Fi drivers.

  For information about setting up a test project, see Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7).
- A validated configuration of the FreeRTOS kernel.

  For information about configuring the FreeRTOS kernel for your platform, see Configuring a FreeRTOS Kernel Port (p. 14).
- Two wireless access points.

Porting

<amazon-freertos>/lib/wifi/portable/*<vendor>*/*<board>*/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.

<amazon-freertos>`/lib/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. In your IDE, under `aws_tests/lib/aws`, create a virtual folder named `wifi`.
2. Add the source file *<amazon-freertos>*`/lib/wifi/portable/`*<vendor>*`/`*<board>*`/aws_wifi.c` to the virtual folder `aws_tests/lib/aws/wifi`.
3. In your IDE, under `aws_tests/application_code/common_tests`, create a virtual folder named `wifi`.
4. Add the source file *<amazon-freertos>*`/tests/common/wifi/aws_test_wifi.c` to the virtual folder `aws_tests/application_code/common_tests/wifi`.

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 `aws_tests/application_code/common_tests/main.c`, and delete the `#if 0` and `#endif` compiler directives in the function definitions of `vApplicationDaemonTaskStartupHook(void)` and `prvWifiConnect(void)`.
2. Open *<amazon-freertos>*`/lib/utils/aws_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 Setting Up the MQTT Library for Testing (p. 51) 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. 24) section, you will be instructed to uncomment this initialization function call.

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

Macro	Value
`clientcredentialWIFI_SSID`	The Wi-Fi SSID as a C string (in quotation marks).
`clientcredentialWIFI_PASSWORD`	The Wi-Fi password as a C string (in quotation marks).
`clientcredentialWIFI_SECURITY`	One of the following: • `eWiFiSecurityOpen` • `eWiFiSecurityWEP` • `eWiFiSecurityWPA` • `eWiFiSecurityWPA2` `eWiFiSecurityWPA2` is recommended.

4. Open *<amazon-freertos>*`/tests/common/include/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: • `eWiFiSecurityOpen` • `eWiFiSecurityWEP` • `eWiFiSecurityWPA` • `eWiFiSecurityWPA2` `eWiFiSecurityWPA2` is recommended.

5. To enable the Wi-Fi tests, open *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/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`
- `WiFiSeperateTasksConnectingAndDisconnectingAtOnce`

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. 24). For information about setting up an echo server, see Setting Up an Echo Server (p. 28).

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...
TEST(Full_WiFi, WiFiOnOff) PASS
TEST(Full_WiFi, WiFiMode) PASS
TEST(Full_WiFi, WiFiConnectionLoop) PASS
TEST(Full_WiFi, WiFiNetworkAddGetDelete) PASS
TEST(Full_WiFi, WiFiPowerManagementMode) PASS
TEST(Full_WiFi, WiFiGetIP) PASS
```

...

```
TEST(Full_WiFi, WIFI_NetworkGet_GetManyNetworks) PASS
TEST(Full_WiFi, WIFI_NetworkAdd_AddManyNetworks) PASS
TEST(Full_WiFi, WIFI_NetworkDelete_DeleteManyNetworks) PASS
TEST(Full_WiFi, WIFI_ConnectAP_ConnectAllChannels) PASS
-----------------------
46 Tests 0 Failures 0 Ignored
OK
----All tests finished----
```

Validation

To officially qualify a device for Amazon 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 Amazon FreeRTOS in the Amazon 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 Amazon FreeRTOS Wi-Fi library to your device, you can start porting a TCP/IP stack. See Porting a TCP/IP Stack (p. 20) for instructions.

Porting a TCP/IP Stack

Amazon 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. 24).

FreeRTOS+TCP is a native TCP/IP stack for the FreeRTOS kernel. FreeRTOS+TCP is maintained by the Amazon FreeRTOS engineering team and is the recommended TCP/IP stack to use with Amazon FreeRTOS. For more information, see Porting FreeRTOS+TCP (p. 20).

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. 23).

> **Note**
> These porting sections only provide instructions for porting to a platform's Ethernet driver. The tests only ensure that the Ethernet 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 that includes the vendor-supplied Ethernet drivers.

  For information about setting up a test project, see Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7).
- A validated configuration of the FreeRTOS kernel.

  For information about configuring the FreeRTOS kernel for your platform, see Configuring a FreeRTOS Kernel Port (p. 14).

Porting

Before you start porting the FreeRTOS-TCP library, check the `<amazon-freertos>/lib/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. Rename the `<board_family>` directory under `<amazon-freertos>/lib/FreeRTOS-Plus-TCP/source/portable/NetworkInterface` to your device's MCU family name.
2. Follow the Porting FreeRTOS+TCP to a Different Microcontroller instructions on FreeRTOS.org to port FreeRTOS+TCP to your device.
3. 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.
4. Implement a new port that uses the vendor-supplied Ethernet drivers in a file called `NetworkInterface.c`, and save the file to `<amazon-freertos>/lib/FreeRTOS-Plus-TCP/source/portable/NetworkInterface/<board_family>`, the directory that you renamed in the first step.

Note
The files in the *<amazon-freertos>*/lib/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 *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/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. In your IDE, under aws_tests/lib, create a virtual folder named FreeRTOS-Plus-TCP.
2. Under aws_tests/lib/FreeRTOS-Plus-TCP, create virtual folders named sources, include, and portable.
3. Add the source files in *<amazon-freertos>*/lib/FreeRTOS-Plus-TCP/source to the virtual folder aws_tests/lib/FreeRTOS-Plus-TCP/source.
4. Add the header files in *<amazon-freertos>*/lib/FreeRTOS-Plus-TCP/include to the virtual folder aws_tests/lib/FreeRTOS-Plus-TCP/include.
5. Under aws_tests/lib/FreeRTOS-Plus-TCP/portable, create a virtual folder named NetworkInterface.
6. Add the port source files in *<amazon-freertos>*/lib/FreeRTOS-Plus-TCP/source/portable/NetworkInterface/*<board_family>* to the virtual folder aws_tests/lib/FreeRTOS-Plus-TCP/portable/NetworkInterface.
7. Under aws_tests/lib/FreeRTOS-Plus-TCP/portable, create a virtual folder named BufferManagement.
8. Add *<amazon-freertos>*/lib/FreeRTOS-Plus-TCP/source/portable/BufferManagement/BufferAllocation_2.c to the virtual folder aws_tests/lib/FreeRTOS-Plus-TCP/portable/BufferManagement.

   Note
   FreeRTOS includes five example heap management implementations under *<amazon-freertos>*/lib/FreeRTOS/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 Amazon FreeRTOS demo applications run properly. Do not use a custom heap implementation.
9. Add *<amazon-freertos>*/lib/FreeRTOS-Plus-TCP/include to your compiler's include path.

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 *<amazon-freertos>*/lib/utils/aws_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 Setting Up the MQTT Library for Testing (p. 51) 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. 24) section, you will be instructed to uncomment this initialization function call.
2. Open *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/application_code/main.c, and uncomment the call to FreeRTOS_IPInit().
3. Fill the following arrays with valid values for your network:

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

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

```
if (eNetworkEvent == eNetworkUp)
{
    configPRINT("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 driver successfully connected to the network, and the test is complete.

Validation

To officially qualify a device for Amazon 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 Amazon FreeRTOS in the Amazon 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 2.0.3.

Prerequisites

To port the lwIP stack, you need the following:

- An IDE project that includes vendor-supplied Ethernet 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. 14).

Porting

Before you port the lwIP TCP/IP stack to your device, check the *<amazon-freertos>*`/lib/third_party/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 *<amazon-freertos>*`/lib/third_party/lwip/src/portable`, create a directory named *<vendor>*`/`*<board>*`/netif`, where the *<vendor>* and *<board>* directories match your platform.
2. Add the *<amazon-freertos>*`/lib/third_pary/lwip/src/netif/ethernetif.c` stub file to `aws_tests/lib/third_party/lwip/src/portable/`*<vendor>*`/`*<board>*`/netif`, and port the 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 *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/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. In your IDE, under `aws_tests/lib/third_party`, create a virtual folder named `lwip`, and then under `aws_tests/lib/third_party/lwip`, create a virtual folder named `src`.
2. Under `aws_tests/lib/third_party/lwip/src`, create a virtual folder named `portable`, and then under `aws_tests/lib/third_party/lwip/src/portable`, create a virtual folder named `arch`.
3. Add the source files in *<amazon-freertos>*`/lib/third_party/lwip/src` to the virtual folder `aws_tests/lib/third_party/lwip/src`.
4. Add the source files in *<amazon-freertos>*`/lib/third_party/lwip/src/portable/arch` to the virtual folder `aws_tests/lib/third_party/lwip/src/portable/arch`.
5. Add the files and directories from *<amazon-freertos>*`/lib/third_party/lwip/src/portable/`*<vendor>*`/`*<board>* to the `aws_tests/lib/third_party/lwip/src/portable/`*<vendor>*`/`*<board>* virtual folder.

   Note
   If you added a `.c` file to `aws_tests/lib/third_party/lwip/src`, and then edited that `.c` file for a port, you must replace the original `.c` file with the edited one in the `aws_tests/lib/third_party/lwip/src` virtual folder.
6. Add the following paths to your compiler's include path:

   - *<amazon-freertos>*`/lib/third_party/lwip/src/include`
   - *<amazon-freertos>*`/lib/third_party/lwip/src/portable`
   - *<amazon-freertos>*`/lib/third_party/lwip/src/portable/`*<vendor>*`/`*<board>*`/include`

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. 29). 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 Amazon 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 Amazon FreeRTOS Secure Sockets Library in the Amazon FreeRTOS User Guide.

Amazon 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 Amazon 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. 16).
- A port of a TCP/IP stack.

  For information about porting a TCP/IP stack, see Porting a TCP/IP Stack (p. 20).
- An echo server.

  Amazon FreeRTOS includes an echo server, written in Go, in the *<amazon-freertos>*`/tools/echo_server` directory. For more information, see Setting Up an Echo Server (p. 28).

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 *<amazon-freertos>*`/lib/secure_sockets/portable/`*<vendor>*`/`*<board>*`/aws_secure_sockets.c`. *<amazon-freertos>*`/lib/include/aws_secure_sockets.h` contains the information required to create the implementations.

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. In your IDE, create a virtual folder named `secure_sockets` under `aws_tests/lib/aws`.
2. If you are using the FreeRTOS+TCP TCP/IP stack, add *<amazon-freertos>*`/lib/secure_sockets/portable/freertos_plus_tcp/aws_secure_sockets.c` to the `aws_tests/lib/aws/secure_sockets` virtual folder.

   If you are using the lwIP TCP/IP stack, add *<amazon-freertos>*`/lib/secure_sockets/portable/lwip/aws_secure_sockets.c` to the `aws_tests/lib/aws/secure_sockets` virtual folder.

   If you are using your own TCP/IP port, add *<amazon-freertos>*`/lib/secure_sockets/portable/`*<vendor>*`/`*<board>*`/aws_secure_sockets.c` to the `aws_tests/lib/aws/secure_sockets` virtual folder.
3. Add *<amazon-freertos>*`/tests/common/secure_sockets/portable/`*<vendor>*`/`*<board>*`/aws_test_tcp.c` to the `aws_tests/application_code/common_tests/secure_sockets` virtual folder.

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 *<amazon-freertos>*/lib/utils/aws_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 Setting Up the MQTT Library for Testing (p. 51) 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 unsecure echo server (*<amazon-freertos>*/tools/echo_server/echo_server.go). For instructions on setting up and running an unsecure echo server, see Setting Up the Echo Server (Without TLS) (p. 29).
3. In *<amazon-freertos>*/tests/common/include/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 *<amazon-freertos>*/tests/common/include/aws_test_tcp.h:

Macro	Value
tcptestECHO_SERVER_ADDR0	192
tcptestECHO_SERVER_ADDR1	168
tcptestECHO_SERVER_ADDR2	2
tcptestECHO_SERVER_ADDR3	6
4. Open *<amazon-freertos>*/tests/common/include/aws_test_tcp.h, and set the tcptestSECURE_SERVER macro to 0 to run the Secure Sockets tests without TLS.
5. Open *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/config_files/aws_test_runner.config.h, and set the testrunnerFULL_TCP_ENABLED macro to 1 to enable the sockets tests.
6. Open *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/application_code/main.c, and 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. 33).

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. 28).
2. Set the IP address and port in *<amazon-freertos>*/tests/common/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 *<amazon-freertos>*`/tests/common/include/aws_test_tcp.h`:

Macro	Value
`tcptestECHO_SERVER_TLS_ADDR0`	`192`
`tcptestECHO_SERVER_TLS_ADDR1`	`168`
`tcptestECHO_SERVER_TLS_ADDR2`	`2`
`tcptestECHO_SERVER_TLS_ADDR3`	`6`
`tcptestECHO_PORT_TLS`	`9000`

3. Open *<amazon-freertos>*`/tests/common/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 *<amazon-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 *<amazon-freertos>*`/tests/common/include/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 *<amazon-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 *<amazon-freertos>*`/tools/certificate_configuration/PEMfileToCString.html` formatting tool.
11. Before you build and run the test project on your device, open *<amazon-freertos>*`/demos/common/include/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 Amazon FreeRTOS. One server uses TLS for secure communication, and the other is unsecured. The servers are located in the *<amazon-freertos>*/`tools/echo_server` folder. The following topics walk you through setting up the echo servers.

Topics

Setting Up the TLS Echo Server

A TLS echo server is defined in *<amazon-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. You can also use a package manager to install OpenSSL for Linux and macOS.

Setting Up the Server

1. Copy *<amazon-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 *<amazon-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 unsecure echo server is defined in *<amazon-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 *<amazon-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 Amazon 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 Amazon FreeRTOS in the Amazon 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 Amazon FreeRTOS Secure Sockets library to your device, you can start porting the PKCS #11 library. See Porting the PKCS #11 Library (p. 29) for instructions.

Porting the PKCS #11 Library

Amazon 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 Amazon FreeRTOS, you need to port the PKCS #11 API to the hardware. For information about the Amazon FreeRTOS PKCS #11 library, see Amazon FreeRTOS PKCS #11 Library in the Amazon FreeRTOS User Guide.

Prerequisites

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

- An IDE project that includes vendor-supplied drivers that are suitable for sensitive data.

  For information about setting up a test project, see Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7).
- A validated configuration of the FreeRTOS kernel.

  For information about configuring the FreeRTOS kernel for your platform, see Configuring a FreeRTOS Kernel Port (p. 14).

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 Amazon FreeRTOS implementation of PKCS #11 uses the cryptographic primitives implemented in the mbedTLS library. Amazon 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 Amazon 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.

<amazon-freertos>/lib/pkcs11/portable/*<vendor>*/*<board>*/aws_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. Use a National Institute of Standards and Technology (NIST)-approved method to provide randomness 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), 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 *<amazon-freertos>*/lib/pkcs11/portable/*<vendor>*/*<board>*/aws_pkcs11_pal.c.

   If your ports use the mbedTLS library for underlying cryptographic and TLS support, but your device does not have a TRNG, do the following:

   a. In the mbedTLS config.h, uncomment MBEDTLS_ENTROPY_NV_SEED, and comment out MBEDTLS_ENTROPY_HARDWARE_ALT.

   b. 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.

   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. In your IDE, create a virtual folder named `pkcs11` under `aws_tests/lib/aws`.
2. Add the source file *<amazon-freertos>*`/lib/pkcs11/portable/`*<vendor>*`/`*<board>*`/aws_pkcs11_pal.c` to the virtual folder `aws_tests/lib/aws/pkcs11`.
3. Create a virtual folder named `pkcs11` under `aws_tests/lib/third_party`.
4. Add the PKCS #11 library headers files from *<amazon-freertos>*`/lib/third_party/pkcs11` to the virtual folder `aws_tests/lib/third_party/pkcs11`.
5. Create a virtual folder named `pkcs11` under `aws_tests/application_code/common_tests`.
6. Add the source file *<amazon-freertos>*`/tests/common/pkcs11/aws_test_pkcs11.c` to the virtual folder `aws_tests/application_code/common_tests/pkcs11`. This file includes the PKCS # 11 tests.
7. Add the source file *<amazon-freertos>*`/lib/pkcs11/mbedtls/aws_pkcs11_mbedtls.c` to the `aws_tests/lib/pkcs11` virtual folder. This file implements PKCS #11 for mbedTLS.
8. Create a virtual folder named `crypto` under `aws_tests/lib`.
9. Import the source file *<amazon-freertos>*`lib/crypto/aws_crypto.c` to the `aws_tests/lib/crypto` virtual folder. This file implements the CRYPTO abstraction wrapper for mbedTLS.
10. Create a virtual folder named `mbedtls` under `aws_tests/lib/third_party`, and then under `aws_tests/lib/third_party`, create virtual folders named `source` and `include`.
11. Add all of the source files from *<amazon-freertos>*`/lib/third_party/mbedtls/library` to the `aws_tests/lib/third_party/mbedtls/source` virtual folder.
12. Add all of the header files from *<amazon-freertos>*`/lib/third_party/mbedtls/include/mbedtls` to the `aws_tests/lib/third_party/mbedtls/include` virtual folder.
13. Add *<amazon-freertos>*`/lib/third_party/mbedtls/include` and *<amazon-freertos>*`/lib/third_party/pkcs11` to the compiler's include path.

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 *<amazon-freertos>*`/lib/utils/aws_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 Setting Up the MQTT Library for Testing (p. 51) 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 *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/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, Digest) PASS

TEST(Full_PKCS11, Digest_ErrorConditions) PASS

TEST(Full_PKCS11, GetFunctionListInvalidParams) PASS

TEST(Full_PKCS11, InitializeFinalizeInvalidParams) PASS
```

...

```
TEST(Full_PKCS11, SignVerifyCryptoApiInteropRSA) PASS

TEST(Full_PKCS11, SignVerifyRoundTrip_MultitaskLoop)d:\treadstonetest_custom\treadstone\tests\common\pkcs11
\aws_test_pkcs11.c:2728::FAIL: This test leaks!

TEST(Full_PKCS11, KeyGenerationEcdsaHappyPath) PASS



-----------------------

33 Tests 1 Failures 0 Ignored

FAIL

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

Testing is complete when all tests pass.

Validation

To officially qualify a device for Amazon 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 Amazon FreeRTOS in the Amazon 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 Amazon FreeRTOS PKCS #11 library to your device, you can start porting the TLS library. See Porting the TLS Library (p. 33) for instructions.

Porting the TLS Library

For TLS authentication, Amazon FreeRTOS uses either mbedTLS or an off-chip TLS implementation, such as those found on some network co-processors. Amazon 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 Amazon 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 Amazon FreeRTOS TLS library, you need the following:

- A port of the Amazon FreeRTOS Secure Sockets library.

  For information about porting the Secure Sockets library to your platform, see Porting the Secure Sockets Library (p. 24).
- A port of the Amazon FreeRTOS PKCS #11 library.

  For information about porting the PKCS #11 library to your platform, see Porting the PKCS #11 Library (p. 29).
- 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
`TLS_Init`	Initialize the TLS context.
`TLS_Connect`	Negotiate TLS and connect to the server.
`TLS_Recv`	Read the requested number of bytes from the TLS connection.
`TLS_Send`	Write the requested number of bytes to the TLS connection.
`TLS_Cleanup`	Free resources consumed by the TLS context.

<amazon-freertos>`/lib/include/aws_tls.h` contains the information required to implement these functions. Save the file in which you implement the functions as *<amazon-freertos>*`/lib/tls/portable/`*<vendor>*`/`*<board>*`/aws_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 *<amazon-freertos>*/demos/common/include/aws_clientcredential.h header file so your device can connect to AWS IoT.

To configure `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 *<1234567890123>*-ats.iot.*<us-east-1>*.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose **Manage**, and then choose **Things**.

   Your device should have an AWS IoT thing name. Make a note of this name.
4. In your IDE, open *<amazon-freertos>*\demos\common\include\aws_clientcredential.h 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 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.

Amazon FreeRTOS is a C language project. You must format certificates and keys before you add them to the aws_clientcredential_keys.h header file.

To format the certificate and key for `aws_clientcredential_keys.h`

1. In a browser window, open *<amazon-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 *<amazon-freertos>*\demos\common\include. This overwrites the existing file in the directory.

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

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. 35) 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 aws_test_tls.h header file.

To format the certificate and key for aws_test_tls.h

1. In a browser window, open *<amazon-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 *<amazon-freertos>*/tests/common/aws_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 *<amazon-freertos>*/tests/common/aws_test_tls.h.

In *<amazon-freertos>*/tests/common/aws_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. 35).

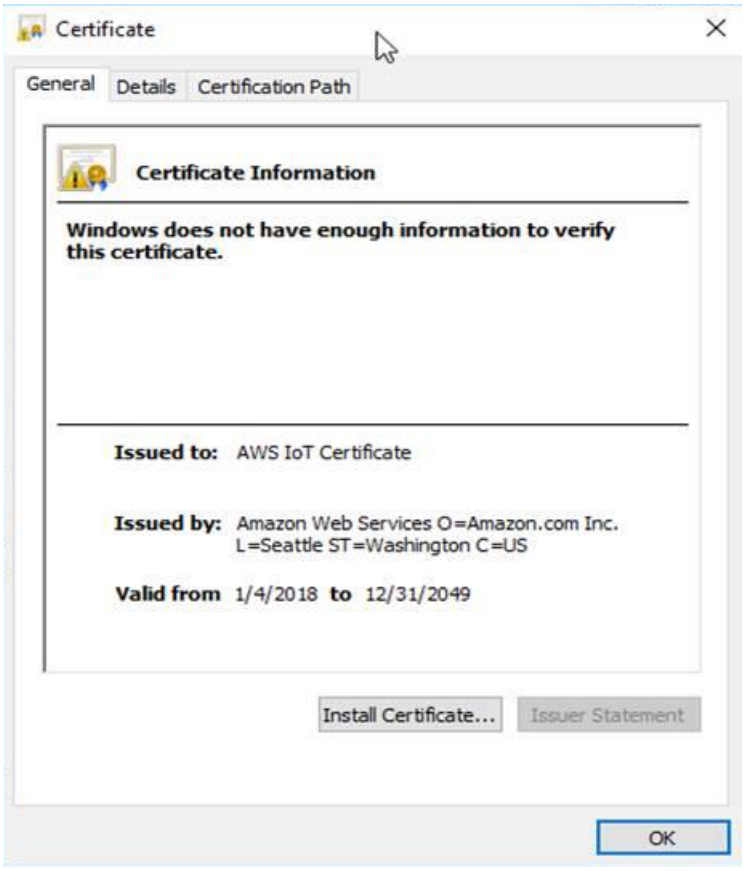
TLS_ConnectMalformedCert()

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
```

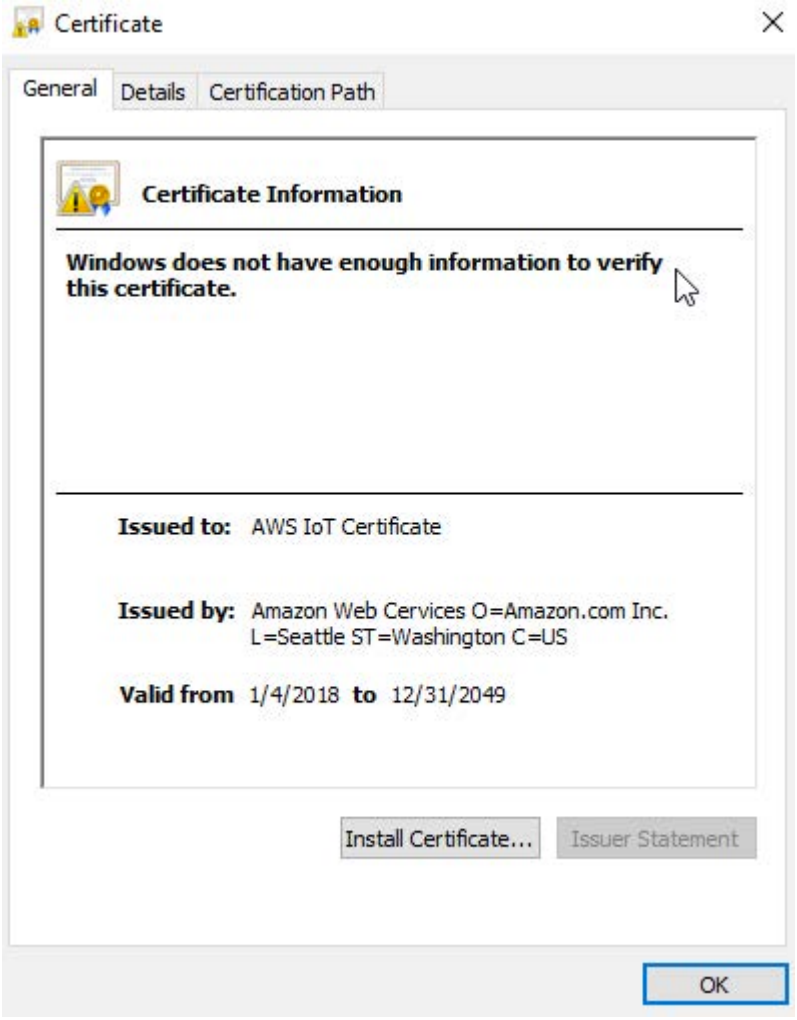
3. Open the DER certificate, and search for the following hexidecimal sequence:

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

   This sequence, translated to plaintext, reads "Amazon Web Services."

4. Change the `53` to `43`, so that the sequence becomes "Amazon Web Cervices" in plaintext, 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 `aws_test_tls.h` header file.

To format the certificate for `aws_test_tls.h`

1. In a browser window, open *<amazon-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 *<amazon-freertos>*`/tests/common/aws_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. 41). 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. 41).
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. 35) 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 `aws_test_tls.h`

1. In a browser window, open *<amazon-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 *<amazon-freertos>*`/tests/common/aws_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 *<amazon-freertos>*`/tests/common/aws_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. 41), but skip the final instructions for registering your device with AWS IoT.

To format the certificate for `aws_test_tls.h`

1. In a browser window, open *<amazon-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 *<amazon-freertos>*`/tests/common/aws_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 *<amazon-freertos>*`/tests/common/aws_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. 43) 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 ctificates 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 extentions 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 extentions 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 an 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, microsecs: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. In your IDE, under `aws_tests/lib/aws`, create a virtual folder named `tls`.
2. If your target hardware does not offload TLS to a separate processor, and you are using mbedTLS, add *<amazon-freertos>*`/lib/tls/aws_tls.c` to the `aws_tests/lib/aws/tls` virtual folder.

   If your target hardware offloads TLS to a separate processor, add *<amazon-freertos>*`/lib/tls/portable/`*<vendor>*`/`*<board>*`/aws_tls.c` to the `aws_tests/lib/aws/tls` virtual folder.
3. Under `aws_tests/application_code/common_tests`, create a virtual folder named `tls`.
4. Add the source file *<amazon-freertos>*`/tests/common/tls/aws_test_tls.c` to the virtual folder `aws_tests/application_code/common_tests/tls`. This file includes the TLS tests.

Setting Up Your Local Testing Environment

There are five separate tests for the TLS port, one for each type of authentication supported by the Amazon 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. 35) to connect your device to AWS IoT.

Each TLS test requires a different certificate/key combination, formatted and defined in either *<amazon-freertos>*`/demos/common/include/aws_clientcredential_keys.h` or *<amazon-freertos>*`/tests/common/aws_test_tls.h`.

Follow the instructions in Setting Up Certificates and Keys for the TLS Tests (p. 36) 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 *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/config_files/aws_test_runner_config.h`, and set the `testrunnerFULL_TLS_ENABLED` macro to `1`.
2. Open *<amazon-freertos>*`/lib/utils/aws_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 Setting Up the MQTT Library for Testing (p. 51) 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. 25) in the Secure Sockets porting documentation.

Validation

To officially qualify a device for Amazon 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 Amazon FreeRTOS in the Amazon 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 Amazon FreeRTOS TLS library to your device, you can start setting up the MQTT library for testing. See Setting Up the MQTT Library for Testing (p. 51) for instructions.

Setting Up 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 Amazon 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 Amazon FreeRTOS MQTT Library in the Amazon FreeRTOS User Guide.

Prerequisites

To set up the Amazon 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. 33).

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

1. In your IDE, under `aws_tests/lib/aws`, create virtual folders named `mqtt` and `bufferpool`.
2. Add all of the source files in *<amazon-freertos>*`/lib/mqtt` to the `aws_tests/lib/aws/mqtt` virtual folder.
3. Add all of the source files in *<amazon-freertos>*`/lib/bufferpool` to the `aws_tests/lib/aws/bufferpool` virtual folder.
4. Under `aws_tests/application_code/common_tests`, create a virtual folder named `mqtt`.
5. Add all of the source files in *<amazon-freertos>*`/tests/common/mqtt` to the virtual folder `aws_tests/application_code/common_tests/mqtt`.

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 *<amazon-freertos>*/lib/utils/aws_system_init.c., and uncomment all of the initialization functions called from SYSTEM_Init().
2. To enable the MQTT tests, open *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/config_files/aws_test_runner_config.h, and set the testrunnerFULL_MQTT_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.

```
Starting Tests...
TEST(Full_MQTT, prvGetTopicFilterType_HappyCases) PASS
TEST(Full_MQTT, prvGetTopicFilterType_ErrorCases) PASS
TEST(Full_MQTT, prvDoesTopicMatchTopicFilter_MatchCases) PASS
TEST(Full_MQTT, prvDoesTopicMatchTopicFilter_NotMatchCases) PASS
TEST(Full_MQTT, MQTT_Init_HappyCase) PASS
TEST(Full_MQTT, MQTT_Init_NULLParams) PASS
TEST(Full_MQTT, MQTT_Connect_HappyCase) PASS
TEST(Full_MQTT, MQTT_Connect_BrokerRejectsConnection) PASS
TEST(Full_MQTT, MQTT_Connect_ConnACKWithoutConnect) PASS
TEST(Full_MQTT, MQTT_Connect_ReservedReturnCodeFromBroker) PASS
TEST(Full_MQTT, MQTT_Connect_ShorterConnACK) PASS
TEST(Full_MQTT, MQTT_Connect_LongerConnACK) PASS
TEST(Full_MQTT, MQTT_Connect_NULLParams) PASS
TEST(Full_MQTT, MQTT_Connect_SecondConnectWhileAlreadyConnected) PASS
TEST(Full_MQTT, MQTT_Connect_SecondConnectWhileWaitingForConnACK) PASS
TEST(Full_MQTT, MQTT_Connect_NetworkSendFailed) PASS

-----------------------
16 Tests 0 Failures 0 Ignored
OK
----All tests finished----
```

If all tests pass, then testing is complete.

Validation

To officially qualify a device for Amazon 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 Amazon FreeRTOS in the Amazon 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 Amazon FreeRTOS MQTT library for your device, you can start porting the OTA agent library. See Porting the OTA Library (p. 53) for instructions.

If your device does not support OTA functionality, you can start porting the Bluetooth Low Energy library. See Porting the BLE Library (p. 57) for instructions.

If your device does not support OTA and BLE functionality, then you are finished porting and can start the Amazon FreeRTOS qualification process. See the Amazon FreeRTOS Qualification Guide for more information.

Porting the OTA Library

With Amazon 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 Amazon FreeRTOS Over-the-Air Updates in the Amazon FreeRTOS User Guide.

You can use the OTA agent library to integrate OTA functionality into your Amazon FreeRTOS applications. For more information, see Amazon FreeRTOS OTA Agent Library in the Amazon FreeRTOS User Guide.

Amazon 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 Amazon 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. 33).
- A bootloader that can support OTA updates.

  For more information about porting a bootloader demo application, see Porting the Bootloader Demo (p. 54).

Porting

<amazon-freertos>/lib/ota/portable/*<vendor>*/*<board>*/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
`prvPAL_Abort`	Aborts an OTA update.
`prvPAL_CreateFileForRx`	Creates a file to store the data chunks as they are received.
`prvPAL_CloseFile`	Closes the specified file. This might authenticate the file if storage that implements cryptographic protection is being used.
`prvPAL_WriteBlock`	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.
`prvPAL_ActivateNewImage`	Activates or launches the new firmware image. For some ports, if the device is programmatically reset synchronously, this function might not return.
`prvPAL_SetPlatformImageState`	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. .
`prvPAL_GetPlatformImageState`	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
`prvPAL_CheckFileSignature`	Verifies the signature of the specified file.
`prvPAL_ReadAndAssumeCertificate`	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 Amazon FreeRTOS, see Porting the Bootloader Demo (p. 54).

Porting the Bootloader Demo

Amazon 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 Amazon 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. To support OTA functionality, the application must meet the following requirements:

- The bootloader is stored in non-volatile memory so it cannot be overwritten.
- The bootloader can verify the cryptographic signature of the downloaded application image.

  The signature verification must be consistent with the OTA image signer.
- The bootloader does not allow rolling back to a previously installed application image.

- The bootloader maintains at least one image that can be booted.
- The bootloader supports self-testing, new OTA images.

  If test execution fails, the bootloader rolls back to the previous valid image. If test execution succeeds, the image is marked as valid and the previous version is erased.
- If the MCU contains more than one image, the newest image is executed.

  The newest version can be determined based on implementation (for example a user defined sequence number, application version, and so on). As per other requirements, this can only be the case until a newer version has been verified and proven functional.
- If the MCU cannot verify any images, it enters a controlled, benign state. In this state, it prevents itself from being taken over by ensuring no actions are performed.

  Note
  These requirements must be met, even in the presence of an accidental or malicious write to any MCU memory location, such as key store, program memory, or RAM.
  The state of the bootloader application must be set by the OTA PAL.

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. In your IDE, create a virtual folder named `ota` under `aws_tests/lib/aws`.
2. Add the source file *<amazon-freertos>*`/lib/ota/portable/`*<vendor>*`/`*<board>*`/aws_ota_pal.c` to the virtual folder `aws_tests/lib/aws/ota`.
3. Create a virtual folder named `ota` under `aws_tests/application_code/common_tests`.
4. Add the following test source files to the virtual folder `aws_tests/application_code/common_tests/ota`:
   - *<amazon-freertos>*`/tests/common/ota/aws_test_cbor.c`
   - *<amazon-freertos>*`/tests/common/ota/aws_test_ota_agent.c`
   - *<amazon-freertos>*`/tests/common/aws_test_pal.c`
   - *<amazon-freertos>*`/demos/common/ota/aws_ota_update_demo.c`

There are two sets of tests for the OTA library port: OTA Agent and OTA PAL Tests (p. 56) and OTA End-to-End Tests (p. 57).

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 *<amazon-freertos>*/tests/*<vendor>*/*<board>*/common/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 *<amazon-freertos>*/tests/ota/test_files. 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 Amazon 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. To enable the end-to-end OTA tests, open *<amazon-freertos>*`/tests/`*<vendor>*`/`*<board>*`/common/config_files/aws_test_runner_config.h`, and set the `testrunner_OTA_END_TO_END_ENABLED` macro to `1`.
2. Follow the setup instructions in the README file (*<amazon-freertos>*`/tools/ota_e2e_test/README.md`).
3. Make sure that running the agent and PAL tests did not modify the `aws_demo_runner.c`, `aws_clientcredential.h`, `aws_clientcredential_keys.h`, `aws_application_version.h`, or `aws_ota_codesigner_certificate.h` header files.
4. To run the OTA end-to-end test script, follow the example in the README file (*<amazon-freertos>*`/tools/ota_e2e_test/README.md`).

Validation

To officially qualify a device for Amazon 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 Amazon FreeRTOS in the Amazon 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 Amazon FreeRTOS OTA library and the bootloader demo, you can start porting the Bluetooth Low Energy library. For instructions, see Porting the BLE Library (p. 57).

If your device does not support BLE functionality, then you are finished and can start the Amazon FreeRTOS qualification process. For more information, see the Amazon FreeRTOS Qualification Guide.

Porting the BLE Library

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

Note
A port of the Amazon FreeRTOS BLE library is currently not required for qualification.

Prerequisites

To port the BLE library, you need the following:

- An IDE project that includes the vendor-supplied BLE drivers.

  For information about setting up a test project, see Setting Up Your Amazon FreeRTOS Source Code for Porting (p. 7).
- A validated configuration of the FreeRTOS kernel.

  For information about configuring the FreeRTOS kernel for your platform, see Configuring a FreeRTOS Kernel Port (p. 14).
- A Raspberry Pi 3 Model B+, with a memory card.

Porting

Three files in the *<amazon-freertos>*`/lib/include/bluetooth_low_energy` folder define the Amazon FreeRTOS BLE 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 expect `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

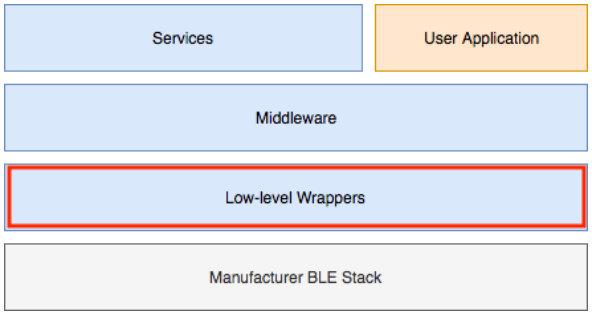
- `pxCharacteristicAddedCb`
- `pxIncludedServiceAddedCb`
- `pxServiceAddedCb`
- `pxConnectionCb`
- `pxUnregisterServerCb`
- `pxRegisterServerCb`

Testing

This diagram shows the BLE testing framework.

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

The BLE porting and qualification tests target the low-level wrapper layer that lies just above the manufacturer's hardware stack in the Amazon FreeRTOS BLE 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 BLE library in the IDE project

1. In your IDE, under `aws_tests/lib/aws`, create a virtual folder named `bluetooth_low_energy`.
2. Add all of the files in *<amazon-freertos>*`/lib/bluetooth_low_energy/portable/`*<vendor>*`/`*<board>* to the virtual folder `aws_tests/lib/aws/bluetooth_low_energy`.
3. Add all of the files in the *<amazon-freertos>*`/lib/include/bluetooth_low_energy` directory to the `aws_tests/lib/aws/include` virtual folder.
4. Under `aws_tests/application_code/common_tests`, create a virtual folder named `ble`.
5. Add the source file *<amazon-freertos>*`/tests/common/aws_test_ble.c` to the virtual folder `aws_tests/application_code/common_tests/ble`.
6. Open `aws_tests/application_code/common_tests/main.c`, and enable the required BLE drivers.

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 *<amazon-freertos>*`/tests/common/framework/bleTestsScripts/runPI.sh` script, and change the IP addresses in the first two lines to the IP address of your Raspberry Pi:

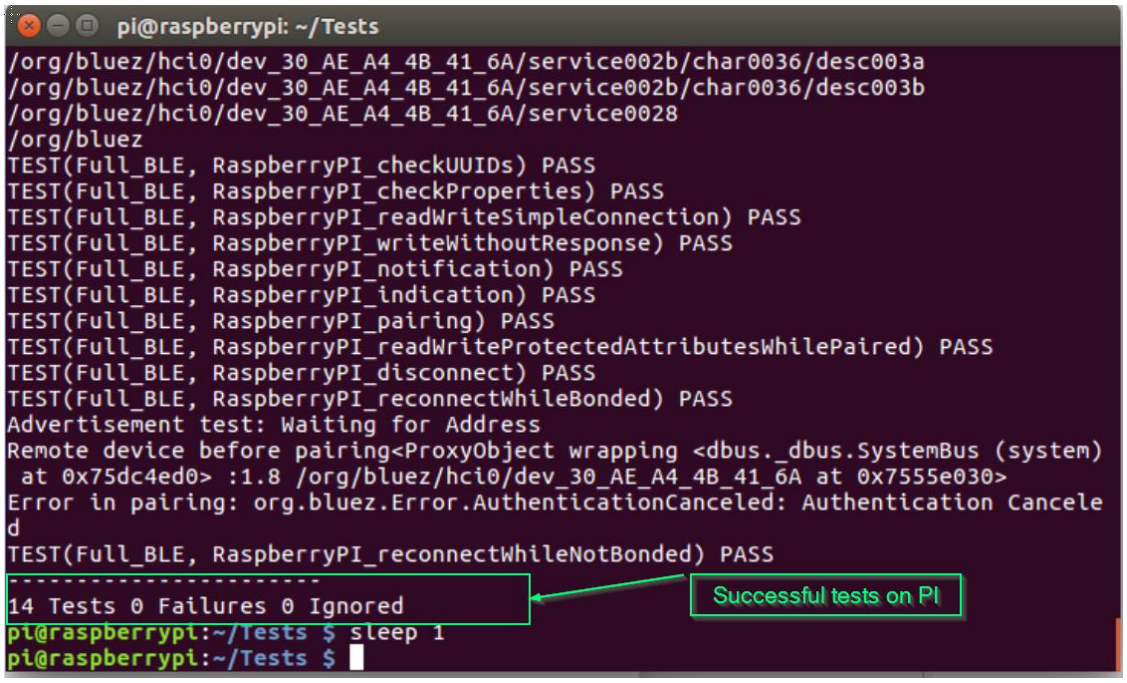
```
#!/bin/sh

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 BLE 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 Amazon 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 Amazon FreeRTOS in the Amazon 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 BLE library, you can start the Amazon FreeRTOS qualification process. For more information, see the Amazon FreeRTOS Qualification Guide.