亚马逊云科技

Porting Guide

FreeRTOS

FreeRTOS: Porting Guide

Table of Contents

FreeRTOS Porting

What is FreeRTOS

Developed in partnership with the world's leading chip companies over a 20-year period, and now downloaded every 170 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use. FreeRTOS includes libraries for connectivity, security, and over-the-air (OTA) updates, and demo applications that demonstrate FreeRTOS features on qualified boards.

For more information, visit FreeRTOS.org.

Porting FreeRTOS to your IoT board

You will need to port FreeRTOS software libraries to your microcontroller-based board based on its features and your application.

To port FreeRTOS to your device

1. Follow the instructions in Downloading FreeRTOS for Porting to download the latest version of FreeRTOS for porting.
2. Follow the instructions in Setting up your workspace and project for porting to configure the files and folders in your FreeRTOS download for porting and testing.
3. Follow the instructions in Porting the FreeRTOS libraries to port the FreeRTOS libraries to your device. Each porting topic includes instructions on testing the ports.

Porting FAQs

What is a FreeRTOS port?

A FreeRTOS port is a board-specific implementation of APIs for the required FreeRTOS libraries and the FreeRTOS kernel 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.

If you have questions about porting that are not answered on this page or in the rest of the FreeRTOS Porting Guide, please see the available FreeRTOS support options.

Downloading FreeRTOS for Porting

Download the latest FreeRTOS or Long Term Support (LTS) version from freertos.org or clone from GitHub (FreeRTOS-LTS) or (FreeRTOS).

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

Alternatively, submodule the individual libraries from the FreeRTOS or FreeRTOS-LTS repository. However, ensure that the library versions match the combination listed in the `manifest.yml` file in the FreeRTOS or FreeRTOS-LTS repository.

After you download or clone FreeRTOS, you can start porting the FreeRTOS libraries to your board. For instructions, see Setting up your workspace and project for porting, and then see Porting the FreeRTOS libraries.

Setting up your workspace and project for porting

Follow the steps below to set up your workspace and project:

- Use a project structure and build system of your choice to import the FreeRTOS libraries.
- Create a project using an Integrated Development Environment (IDE) and toolchain supported by your board.
- Include the board support packages (BSP) and board-specific drivers in your project.

Once your workspace is set up, you can start porting individual FreeRTOS libraries.

Porting the FreeRTOS libraries

Before you start porting, follow the instructions at Setting up your workspace and project for porting.

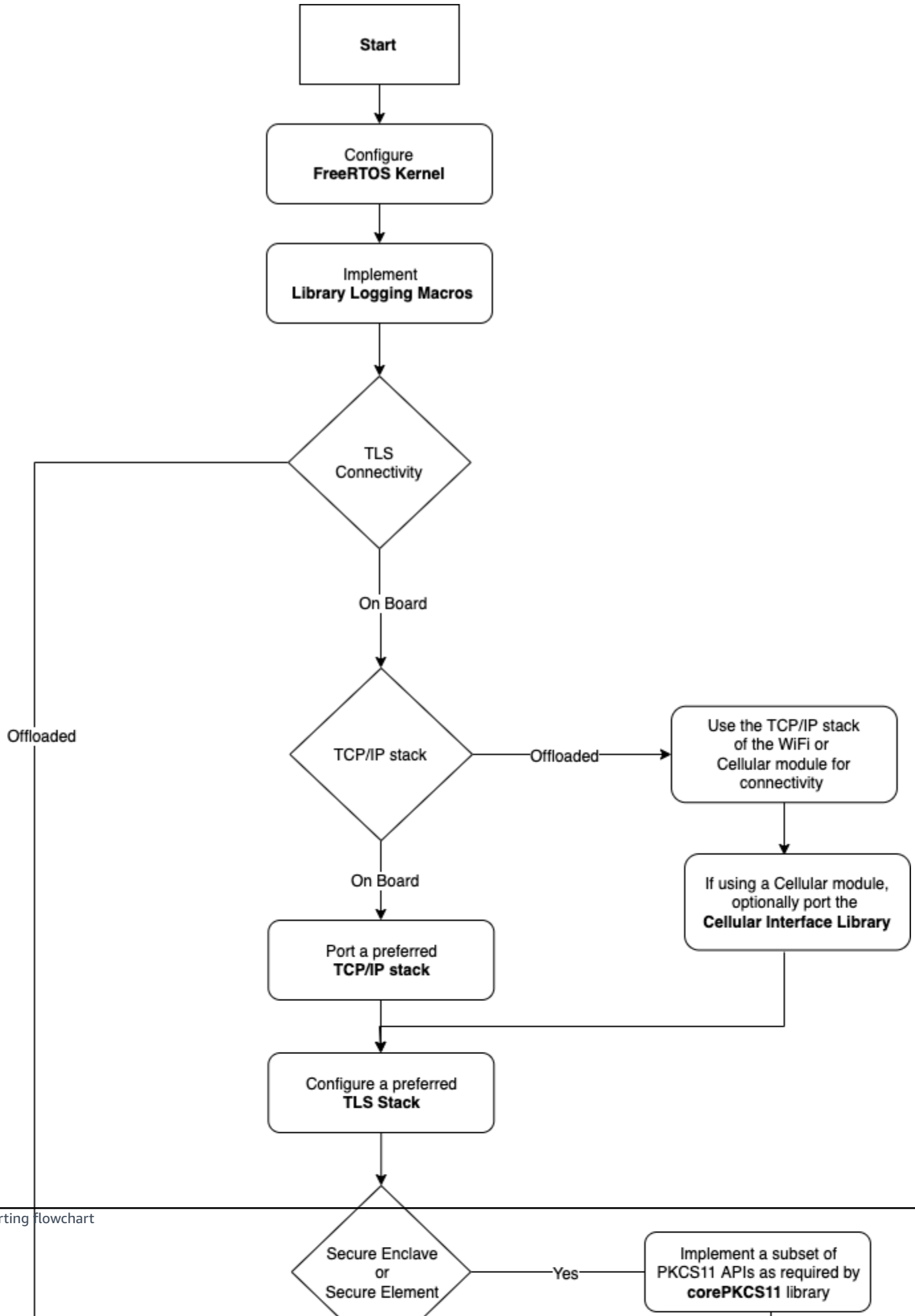
The FreeRTOS porting flowchart describes the libraries required for porting.

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

1. Configuring a FreeRTOS kernel port
2. Implementing the library logging macros
3. Porting a TCP/IP stack
4. Porting the Network Transport Interface
5. Porting the corePKCS11 library
6. Configuring the coreMQTT library
7. Configuring the coreHTTP library
8. Porting the Amazon IoT over-the-air (OTA) update library
9. Porting the Cellular Interface library

FreeRTOS porting flowchart

Use the porting flowchart below as a visual aid, as you port FreeRTOS to your board.



Porting flowchart

Configuring a FreeRTOS kernel port

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

FreeRTOS uses the FreeRTOS kernel for multitasking and intertask communications. For more information, see the FreeRTOS kernel fundamentals in the *FreeRTOS User Guide* and FreeRTOS.org.

> **Note**
>
> Porting the FreeRTOS kernel to a new architecture is not included in this documentation. If you are interested, contact the FreeRTOS engineering team.
> For the FreeRTOS Qualification program, only existing FreeRTOS kernel ports are supported. Modifications to these ports are not accepted within the program. Review the FreeRTOS kernel port policy for more information.

Prerequisites

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

- An official FreeRTOS kernel port, or FreeRTOS supported ports 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 workspace and project for porting.

Configuring the FreeRTOS kernel

FreeRTOS kernel is customized using a configuration file called `FreeRTOSConfig.h`. This file specifies application-specific configuration settings for the kernel. For a description of each configuration option, see Customization on FreeRTOS.org.

To configure the FreeRTOS kernel to work with your device, include `FreeRTOSConfig.h`, and modify any additional FreeRTOS configurations.

For a description of each configuration option, see Customization configurations on FreeRTOS.org.

Testing

- Run a simple FreeRTOS task to log a message to serial output console.
- Verify that the message outputs to console as expected.

Implementing the library logging macros

The FreeRTOS libraries use the following logging macros, listed in increasing order of verbosity.

- `LogError`
- `LogWarn`
- `LogInfo`
- `LogDebug`

A definition for all the macros must be provided. The recommendations are:

- Macros should support `C89` style logging.
- Logging should be thread safe. Log lines from multiple tasks must not interleave with each other.
- Logging APIs must not block, and must free application tasks from blocking on I/O.

Refer to the Logging Functionality on FreeRTOS.org for implementation specifics. You can see an implementation in this example.

Testing

- Run a test with multiple tasks to verify logs do not interleave.
- Run a test to verify that the logging APIs do not block on I/O.
- Test logging macros with various standards, such as `C89`, `C99` style logging.
- Test logging macros by setting different log levels, such as `Debug`, `Info`, `Error`, and `Warning`.

Porting a TCP/IP stack

This section provides instruction for porting and testing on-board TCP/IP stacks. If your platform offloads TCP/IP and TLS functionality to a separate network processor or module, you can skip this porting section and visit Porting the Network Transport Interface.

FreeRTOS+TCP is a native TCP/IP stack for the FreeRTOS kernel. FreeRTOS+TCP is developed and maintained by the FreeRTOS engineering team and is the recommended TCP/IP stack to use with FreeRTOS. For more information, see Porting FreeRTOS+TCP. Alternatively, you can use the third-party TCP/IP stack lwIP . The testing instruction provided in this section uses the transport interface tests for TCP plain text, and is not dependent on the specific implemented TCP/IP stack.

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 or Wi-Fi drivers.

  For information about setting up a test project, see Setting up your workspace and project for porting.
- A validated configuration of the FreeRTOS kernel.

  For information about configuring the FreeRTOS kernel for your platform, see Configuring a FreeRTOS kernel port.

Porting

Before you start porting the FreeRTOS+TCP library, check the GitHub directory to see if a port to your board already exists.

If a port does not exist, do the following:

1. Follow the Porting FreeRTOS+TCP to a Different Microcontroller instructions on FreeRTOS.org to port FreeRTOS+TCP to your device.

2. If necessary, follow the Porting FreeRTOS+TCP to a New Embedded C Compiler instructions on FreeRTOS.org to port FreeRTOS+TCP to a new compiler.
3. Implement a new port that uses the vendor-supplied Ethernet or Wi-Fi drivers in a file called `NetworkInterface.c`. Visit the GitHub repository for a template.

After you create a port, or if a port already exists, create `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

Whether you use FreeRTOS+TCP library or a third party library, follow the steps below for testing:

- Provide an implementation for `connect/disconnect/send/receive` APIs in transport interface tests.
- Setup an echo server in plain text TCP connection mode, and run transport interface tests.

> **Note**
>
> To officially qualify a device for FreeRTOS, if your architecture requires to port a TCP/IP software stack, you need to validate the device's ported source code against transport interface tests in plain text TCP connection mode with Amazon IoT Device Tester. Follow the instructions in Using Amazon IoT Device Tester for FreeRTOS in the *FreeRTOS User Guide* to set up Amazon IoT 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 the corePKCS11 library

The Public Key Cryptography Standard #11 defines a platform-independent API to manage and use cryptographic tokens. PKCS 11 refers to the standard and the APIs defined by it. The PKCS #11 cryptographic API abstracts key storage, get/set properties for cryptographic objects, and session semantics. It's widely used for manipulating common cryptographic objects. Its functions allow application software to use, create, modify, and delete cryptographic objects, without exposing those objects to the application's memory.

FreeRTOS libraries and reference integrations use a subset of the PCKS #11 interface standard, with a focus on the operations involving asymmetric keys, random number generation, and hashing. The below table lists the use cases and required PKCS #11 APIs to support.

Use Cases

Use Case	Required PKCS #11 API Family
All	Initialize, Finalize, Open/Close Session, GetSlotList, Login
Provisioning	GenerateKeyPair, CreateObject, DestroyObject, InitToken, GetTokenInfo
TLS	Random, Sign, FindObject, GetAttributeValue
FreeRTOS+TCP	Random, Digest
OTA	Verify, Digest, FindObject, GetAttributeValue

When to implement a complete PKCS #11 module

Storing private keys in general-purpose flash memory can be convenient in evaluation and rapid prototyping scenarios. We recommend you use dedicated cryptographic hardware to reduce the threats of data theft and device duplication in production scenarios. Cryptographic hardware includes components with features that prevent cryptographic secret keys from being exported. To support this, you will have to implement a subset of PKCS #11 required to work with FreeRTOS libraries as defined in the above table.

When to use FreeRTOS corePKCS11

The corePKCS11 library contains a software-based implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. This is provided for rapid prototyping and evaluation scenarios where the hardware does not have a dedicated cryptographic hardware. In this case, you only have to implement corePKCS11 PAL to make the corePKCS11 software-based implementation to work with your hardware platform.

Porting corePKCS11

You will have to have implementations to read and write cryptographic objects to non-volatile memory (NVM), such as on-board flash memory. Cryptographic objects must be stored in a section of NVM that is not initialized and is not erased on device reprogramming. Users of the corePKCS11 library will provision devices with credentials, and then reprogram the device with a new application that accesses these credentials through the corePKCS11 interface. The corePKCS11 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 boot-loader and over-the-air (OTA) updates
- A Just-In-Time provisioning certificate

Include the header file and implement the PAL APIs defined.

PAL APIs

Function	Description
PKCS11_PAL_Initialize	Initializes the PAL layer. Called by the corePKCS11 library at the start of its initializ ation sequence.
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.

Function	Description
PKCS11_PAL_GetObjectValueCleanup	Cleanup for the `PKCS11_PAL_GetObjectValue` call. Can be used to free memory allocated in a `PKCS11_PAL_GetObjectValue` call.

Testing

If you use the FreeRTOS corePKCS11 library or implement the required subset of PKCS11 APIs, you must pass FreeRTOS PKCS11 tests. These test if the required functions for FreeRTOS libraries perform as expected.

This section also describes how you can locally run the FreeRTOS PKCS11 tests with the qualification tests.

Prerequisites

To set up the FreeRTOS PKCS11 tests, the following has to be implemented.

- A supported port of PKCS11 APIs.
- An implementation of FreeRTOS qualification tests platform functions which include the following:
  - `FRTest_ThreadCreate`
  - `FRTest_ThreadTimedJoin`
  - `FRTest_MemoryAlloc`
  - `FRTest_MemoryFree`

(See the README.md file for the FreeRTOS Libraries Integration Tests for PKCS #11 on GitHub.)

Porting tests

- Add FreeRTOS-Libraries-Integration-Tests as a submodule into your project. The submodule can be placed in any directory of the project, as long as it can be built.

- Copy `config_template/test_execution_config_template.h` and `config_template/test_param_config_template.h` to a project location in the build path, and rename them to `test_execution_config.h` and `test_param_config.h`.
- Include relevant files into the build system. If using `CMake`, `qualification_test.cmake` and `src/pkcs11_tests.cmake` can be used to include relevant files.
- Implement `UNITY_OUTPUT_CHAR` so that test output logs and device logs do not interleave.
- Integrate the MbedTLS, which verifies the cryptoki operation result.
- Call `RunQualificationTest()` from the application.

Configuring tests

The PKCS11 test suite must be configured according to the PKCS11 implementation. The following table lists the configuration required by PKCS11 tests in the `test_param_config.h` header file.

PKSC11 test configurations

Configuration	Description
PKCS11_TEST_RSA_KEY_SUPPORT	The porting supports RSA key functions.
PKCS11_TEST_EC_KEY_SUPPORT	The porting supports EC key functions.
PKCS11_TEST_IMPORT_PRIVATE_KEY_SUPPORT	The porting supports the import of the private key. RSA and EC key import are validated in the test if the supporting key functions are enabled.
PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT	The porting supports keypair generation. EC keypair generation is validated in the test if the supporting key functions are enabled.
PKCS11_TEST_PREPROVISIONED_SUPPORT	The porting has pre-provisioned credentials. `PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS`, `PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS` and `PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS`, are examples of the credentials.

Configuration	Description
PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS	The label of the private key used in the test.
PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS	The label of the public key used in the test.
PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS	The label of the certificate used in the test.
PKCS11_TEST_JITP_CODEVERIFY_ROOT_CERT_SUPPORTED	The porting supports storage for JITP. Set this to 1 to enable the JITP `codeverify` test.
PKCS11_TEST_LABEL_CODE_VERIFICATION_KEY	The label of the code verification key used in JITP `codeverify` test.
PKCS11_TEST_LABEL_JITP_CERTIFICATE	The label of the JITP certificate used in JITP `codeverify` test.
PKCS11_TEST_LABEL_ROOT_CERTIFICATE	The label of the root certificate used in JITP `codeverify` test.

FreeRTOS libraries and reference integrations must support a minimum of one key function configuration like RSA or Elliptic curve keys, and one key provisioning mechanism supported by the PKCS11 APIs. The test must enable the following configurations:

- At least one of the following key function configurations:
  - PKCS11_TEST_RSA_KEY_SUPPORT
  - PKCS11_TEST_EC_KEY_SUPPORT
- At least one of the following key provisioning configurations:
  - PKCS11_TEST_IMPORT_PRIVATE_KEY_SUPPORT
  - PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT
  - PKCS11_TEST_PREPROVISIONED_SUPPORT

The pre-provisioned device credential test must run under the following conditions:

- `PKCS11_TEST_PREPROVISIONED_SUPPORT` must be enabled and other provisioning mechanisms disabled.
- Only one key function, either `PKCS11_TEST_RSA_KEY_SUPPORT` or `PKCS11_TEST_EC_KEY_SUPPORT`, is enabled.
- Set up the pre-provisioned key labels according to your key function, including `PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS`, `PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS` and `PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS`. These credentials must exist before running the test.

The test may need to run several times with different configurations, if the implementation supports pre-provisioned credentials and other provisioning mechanisms.

> **Note**
>
> The objects with labels `PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS`, `PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS` and `PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS` are destroyed during the test if either `PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT` or `PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT` is enabled.

Running tests

This section describes how you can locally test the PKCS11 interface with the qualification tests. Alternatively, you can also use IDT to automate the execution. See Amazon IoT Device Tester for FreeRTOS in the *FreeRTOS User Guide* for details.

The following instructions describe how to run the tests:

- Open `test_execution_config.h` and define **CORE_PKCS11_TEST_ENABLED** to 1.
- Build and flash the application to your device to run. The test result are output to the serial port.

The following is an example of the output test result.

```
TEST(Full_PKCS11_StartFinish, PKCS11_StartFinish_FirstTest) PASS
TEST(Full_PKCS11_StartFinish, PKCS11_GetFunctionList) PASS
TEST(Full_PKCS11_StartFinish, PKCS11_InitializeFinalize) PASS
TEST(Full_PKCS11_StartFinish, PKCS11_GetSlotList) PASS
TEST(Full_PKCS11_StartFinish, PKCS11_OpenSessionCloseSession) PASS
TEST(Full_PKCS11_Capabilities, PKCS11_Capabilities) PASS
TEST(Full_PKCS11_NoObject, PKCS11_Digest) PASS
TEST(Full_PKCS11_NoObject, PKCS11_Digest_ErrorConditions) PASS
TEST(Full_PKCS11_NoObject, PKCS11_GenerateRandom) PASS
TEST(Full_PKCS11_NoObject, PKCS11_GenerateRandomMultiThread) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_CreateObject) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_FindObject) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_GetAttributeValue) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_Sign) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_FindObjectMultiThread) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_GetAttributeValueMultiThread) PASS
TEST(Full_PKCS11_RSA, PKCS11_RSA_DestroyObject) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_GenerateKeyPair) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_CreateObject) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_FindObject) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_GetAttributeValue) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_Sign) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_Verify) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_FindObjectMultiThread) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_GetAttributeValueMultiThread) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_SignVerifyMultiThread) PASS
TEST(Full_PKCS11_EC, PKCS11_EC_DestroyObject) PASS

-----------------------
27 Tests 0 Failures 0 Ignored
OK
```

Testing is complete when all tests pass.

> **Note**
>
> To officially qualify a device for FreeRTOS, you must validate the device's ported source code with Amazon IoT Device Tester. Follow the instructions in Using Amazon IoT Device Tester for FreeRTOS in the FreeRTOS User Guide to set up Amazon IoT 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 Amazon IoT Device Tester `configs` folder.

Porting the Network Transport Interface

Integrating the TLS library

For Transport Layer Security (TLS) authentication, use your preferred TLS stack. We recommend using Mbed TLS because it is tested with FreeRTOS libraries. You can find an example of this at this GitHub repository.

Regardless of the TLS implementation used by your device, you must implement the underlying transport hooks for TLS stack with TCP/IP stack. They must support the TLS cipher suites that are supported by Amazon IoT.

Porting the Network Transport Interface library

You must implement a network transport interface to use coreMQTT and coreHTTP. Network Transport Interface contains function pointers and context data required to send and receive data on a single network connection. See Transport Interface for more details. FreeRTOS provides a set of built-in network transport interface tests to validate these implementations. The following section guides you how to set up your project to run these tests.

Prerequisites

To port this test, you need the following:

- A project with a build system that can build FreeRTOS with a validated FreeRTOS kernel port.
- Working implementation of network drivers.

Porting

- Add FreeRTOS-Libraries-Integration-Tests as a submodule into your project. It doesn't matter where the submodule is placed in the project, as long as it can be built.
- Copy `config_template/test_execution_config_template.h` and `config_template/test_param_config_template.h` to a project location in the build path, and rename them to `test_execution_config.h` and `test_param_config.h`.
- Include relevant files into the build system. If using `CMake`, `qualification_test.cmake` and `src/transport_interface_tests.cmake` are used to include relevant files.
- Implement the following functions at an appropriate project location:

- • A `network connect function`: The signature is defined by `NetworkConnectFunc` in `src/common/network_connection.h`. This function takes in a pointer to network context, a pointer to host info, and a pointer to network credentials. It establishes a connection with the server specified in the host info with the provided network credentials.
  - A `network disconnect function`: The signature is defined by `NetworkDisconnectFunc` in `src/common/network_connection.h`. This function takes in a pointer to a network context. It disconnects a previously established connection stored in the network context.
  - `setupTransportInterfaceTestParam()`: This is defined in `src/transport_interface/transport_interface_tests.h`. The implementation must have exactly the same name and signature as defined in `transport_interface_tests.h`. This function takes in a pointer to a *TransportInterfaceTestParam* struct. It will populate the fields in the *TransportInterfaceTestParam* struct that is used by the transport interface test.
- Implement **UNITY_OUTPUT_CHAR** so that test output logs do not interleave with device logs.
- Call `runQualificationTest()`from the application. The device hardware must be properly initialized and the network must be connected before the call.

Credential management (on-device generated key)

When **FORCE_GENERATE_NEW_KEY_PAIR** in `test_param_config.h` is set to 1, the device application generates a new on-device key pair and outputs the public key. The device application uses **ECHO_SERVER_ROOT_CA** and **TRANSPORT_CLIENT_CERTIFICATE** as the echo server root CA and client certificate when establishing a TLS connection with the echo server. IDT sets these parameters during the qualification run.

Credential Management (importing key)

The device application uses **ECHO_SERVER_ROOT_CA**, **TRANSPORT_CLIENT_CERTIFICATE** and **TRANSPORT_CLIENT_PRIVATE_KEY** in `test_param_config.h` as the echo server root CA, client certificate, and client private key when establishing a TLS connection with the echo server. IDT sets these parameters during the qualification run.

Testing

This section describes how you can locally test the transport interface with the qualification tests. Additional details can be found in the README.md file provided in the transport_interface section of the FreeRTOS-Libraries-Integration-Tests on GitHub.

Alternatively, you can also use IDT to automate the execution. See Amazon IoT Device Tester for FreeRTOS in the *FreeRTOS User Guide* for details.

Enable the test

Open `test_execution_config.h` and define **TRANSPORT_INTERFACE_TEST_ENABLED** to 1.

Set up the echo server for testing

An echo server accessible from the device running the tests is required for local testing. The echo server must support TLS if the transport interface implementation supports TLS. If you don't have one already, FreeRTOS-Libraries-Integration-Tests GitHub repository has an echo server implementation.

Configuring the project for testing

In `test_param_config.h`, update **ECHO_SERVER_ENDPOINT** and **ECHO_SERVER_PORT** to the endpoint and server setup in the previous step.

Setup credentials (on-device generated key)

- Set **ECHO_SERVER_ROOT_CA** to the server certificate of the echo server.
- Set **FORCE_GENERATE_NEW_KEY_PAIR** to 1 to generate a key pair and get the public key.
- Set **FORCE_GENERATE_NEW_KEY_PAIR** back to 0 after key generation.
- User the public key and server key and certificate to generate client certificate.
- Set **TRANSPORT_CLIENT_CERTIFICATE** to the generated client certificate.

Setup credentials (importing key)

- Set **ECHO_SERVER_ROOT_CA** to the server certificate of the echo server.
- Set **TRANSPORT_CLIENT_CERTIFICATE** to the pre-generated client certificate.
- Set **TRANSPORT_CLIENT_PRIVATE_KEY** to the pre-generated client private key.

Build and flash the application

Build and flash the application using the tool-chain of your choice. When `runQualificationTest()` is invoked, the transport interface tests will run. Test results are outputted to the serial port.

> **Note**
>
> To officially qualify a device for FreeRTOS, you must validate the device's ported source code against OTA PAL and OTA E2E test groups with Amazon IoT Device Tester. Follow the instructions in Using Amazon IoT Device Tester for FreeRTOS in the *FreeRTOS User Guide* to set up Amazon IoT 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 Amazon IoT Device Tester `configs` folder.

Configuring the coreMQTT library

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

The coreMQTT library implements the MQTT protocol for devices running FreeRTOS. The coreMQTT library doesn't need to be ported, but your device's test project must pass all MQTT tests for qualification. For more information, see coreMQTT Library in the *FreeRTOS User Guide.*

Prerequisites

To set up the coreMQTT library tests, you need a network transport interface port. See Porting the Network Transport Interface to learn more.

Testing

Run coreMQTT Integration tests:

- Register your client certificate with MQTT broker.
- Set the broker endpoint in `config` and run the integration tests.

Create reference MQTT demo

We recommend using the coreMQTT agent to handle thread safety for all MQTT operations. The user will also need publish and subscribe tasks, and Device Advisor tests to validate if the application integrates TLS, MQTT and other FreeRTOS libraries effectively.

To officially qualify a device for FreeRTOS, validate your integration project with Amazon IoT Device Tester MQTT test cases. See Amazon IoT Device Advisor workflow for instructions to set up and test. Mandated test cases for TLS and MQTT are listed below:

TLS Test Cases

Test Case	Test cases	Required tests
TLS	TLS Connect	Yes
TLS	TLS Support Amazon IoT Cipher Suites	A recommended cipher suite
TLS	TLS Unsecure Server Cert	Yes
TLS	TLS Incorrect Subject Name Servr Cert	Yes

MQTT Test Cases

Test Case	Test cases	Required tests
MQTT	MQTT Connect	Yes
MQTT	MQTT Connect Jitter Retries	Yes without warnings
MQTT	MQTT Subscribe	Yes
MQTT	MQTT Publish	Yes
MQTT	MQTT ClientPuback QoS1	Yes
MQTT	MQTT No Ack PingResp	Yes

Configuring the coreHTTP library

Devices on the edge can use the HTTP protocol to communicate with the Amazon Cloud. Amazon IoT services host an HTTP server that sends and receives messages to and from connected devices at the edge.

Testing

Follow the steps below for testing:

- Setup the PKI for TLS mutual authentication with Amazon or an HTTP server.
- Run CoreHTTP integration tests.

Porting the Amazon IoT over-the-air (OTA) update library

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

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

For more information, see FreeRTOS Over-the-Air Updates in the *FreeRTOS User Guide* along with the Amazon IoT Over-the-air Update Documentation.

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

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

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

Prerequisites

- Complete the instructions in Setting up your workspace and project for porting.
- Create a network transport interface port.

For information, see Porting the Network Transport Interface.

- Integrate coreMQTT library. See coreMQTT library in the FreeRTOS User Guide.
- Create a bootloader that can support OTA updates.

Platform porting

You must provide an implementation of the OTA portable abstraction layer (PAL) to port the OTA library to a new device. The PAL APIs are defined in the ota_platform_interface.h file for which implementation specific details must be provided.

Function name	Description
`otaPal_Abort`	Stops an OTA update.
`otaPal_CreateFileForRx`	Creates a file to store the received data chunks.
`otaPal_CloseFile`	Closes the specified file. This might authenticate the file if you use storage that implements cryptographic protection.
`otaPal_WriteBlock`	Writes a block of data to the specified file at the given offset. On success, the function returns the number of bytes written. Otherwise, the function returns a negative error code. The block size will always be a power of two and will be aligned. For more information, see OTA library configuration.
`otaPal_ActivateNewImage`	Activates or launches the new firmware image. For some ports, if the device is programmatically reset synchronously, this function will not return.
`otaPal_SetPlatformImageState`	Does what is required by the platform to accept or reject the most recent OTA firmware image (or bundle). To implement this function,

Function name	Description
	see the documentation for your board (platform) details and architecture.
`otaPal_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 name	Description
`otaPal_CheckFileSignature`	Verifies the signature of the specified file.
`otaPal_ReadAndAssumeCertificate`	Reads the specified signer certificate from the file system and returns it to the caller.
`otaPal_ResetDevice`	Resets the device.

> **Note**
>
> Make sure that you have a bootloader that can support OTA updates. For instructions on creating your Amazon IoT device bootloader, see IoT device bootloader.

E2E and PAL tests

Run OTA PAL and E2E tests.

E2E tests

OTA end to end (E2E) test is used to verify a device's OTA capability and to simulate scenarios from reality. This test will include error handling.

Prerequisites

To port this test, you need the following:

- A project with an Amazon OTA library integrated in it. Visit the OTA Library Porting Guide for additional information.

- Port the demo application using the OTA library to interact with Amazon IoT Core to do the OTA updates. See Porting the OTA demo application.
- Set up the IDT tool. This runs the OTA E2E host application to build, flash, and monitor the device with different configurations, and validates the OTA library integration.

Porting the OTA demo application

The OTA E2E test must have an OTA demo application to validate the OTA library integration. The demo application must have the capacity to perform OTA firmware updates. You can find the FreeRTOS OTA demo application at FreeRTOS GitHub repository. We recommend that you use the demo application as a reference, and modify it according to your specifications.

Porting steps

1. Initialize the OTA agent.
2. Implement the OTA application callback function.
3. Create the OTA agent event processing task.
4. Start the OTA agent.
5. Monitor the OTA agent statistics.
6. Shut down the OTA agent.

Visit FreeRTOS OTA over MQTT - Entry point of the demo for detailed instructions.

Configuration

The following configurations are necessary to interact with Amazon IoT Core:

- Amazon IoT Core client credentials
  - Set-up **democonfigROOT_CA_PEM** in `Ota_Over_Mqtt_Demo/demo_config.h` with Amazon Trust Services endpoints. See Amazon server-authentication for more details.
  - Set-up **democonfigCLIENT_CERTIFICATE_PEM** and **democonfigCLIENT_PRIVATE_KEY_PEM** in `Ota_Over_Mqtt_Demo/demo_config.h` with your Amazon IoT client credentials. See Amazon client-authentication detailsto learn about client certificates and private keys.
- Application version
- OTA Control Protocol
- OTA Data Protocol

- Code Signing credentials
- Other OTA library configurations

You can find the preceding information in `demo_config.h` and `ota_config.h` in FreeRTOS OTA demo applications. Visit FreeRTOS OTA over MQTT - Setting up the device for more information.

Build verification

Run the demo application to run the OTA job. When it completes successfully, you can continue to run the OTA E2E tests.

FreeRTOS OTA demo provides detailed information about setting up an OTA client and an Amazon IoT Core OTA job on the FreeRTOS windows simulator. Amazon OTA supports both MQTT and HTTP protocols. Refer to the following examples for more details:

- OTA over MQTT Demo on Windows Simulator
- OTA over HTTP Demo on Windows Simulator

Running tests with the IDT tool

To run the OTA E2E tests, you must use Amazon IoT Device Tester (IDT) to automate the execution. See Amazon IoT Device Tester for FreeRTOS in the *FreeRTOS User Guide* for more details.

E2E test cases

Test case	Description
`OTAE2EGreaterVersion`	Happy path test for regular OTA updates. It creates an update with a newer version, which the device updates successfully.
`OTAE2EBackToBackDownloads`	This test creates 3 consecutive OTA updates. The device is expected to update 3 consecutive times.
`OTAE2ERollbackIfUnableToConnectAfterUpdate`	This test verifies that the device rollbacks to the previous firmware if it cannot connect to network with the new firmware.

Test case	Description
`OTAE2ESameVersion`	This test confirms that the device rejects the incoming firmware if the version stays the same.
`OTAE2EUnsignedImage`	This test verifies that the device rejects an update if the image is not signed.
`OTAE2EUntrustedCertificate`	This test verifies that the device rejects an update if the firmware is signed with an untrusted certificate.
`OTAE2EPreviousVersion`	This test verifies that the device rejects an older update version.
`OTAE2EIncorrectSigningAlgorithm`	Different devices support different signing and hashing algorithms. This test verifies that the device fails the OTA update if it's created with a non-supported algorithm.
`OTAE2EDisconnectResume`	This is the happy path test for the suspend and resume feature. This test creates an OTA update and starts the update. It then connects to Amazon IoT Core with the same client ID (thing name) and credentials. Amazon IoT Core then disconnects the device. The device is expected to detect that it is disconnected from Amazon IoT Core, and after a period of time, move itself to a suspended state and try to reconnect to Amazon IoT Core and resume the download.

Test case	Description
`OTAE2EDisconnectCancelUpdate`	This test checks if the device can recover itself if the OTA job gets canceled while it is in a suspended state. It does the same thing as the `OTAE2EDisconnectResume` test, except that after connecting to Amazon IoT Core, which disconnects the device, it cancels the OTA update. A new update is created. The device is expected to reconnect to the Amazon IoT Core, abort the current update, go back to waiting state, and accept and finish the next update.
`OTAE2EPresignedUrlExpired`	When an OTA update is created, you can configure the lifetime of the S3 pre-signed url. This test verifies that the device is able to perform an OTA, even if it cannot finish the download when the url expires. The device is expected to request a new job document, which contains a new url to resume the download.
`OTAE2E2UpdatesCancel1st`	This test creates two OTA updates in a row. When the device reports that it is downloadi ng the first update, the test force-cancels the first update. The device is expected to abort the current update and pick up the second update, and complete it.
`OTAE2ECancelThenUpdate`	This test creates two OTA updates in a row. When the device reports that it is downloadi ng the first update, the test force-cancels the first update. The device is expected to abort the current update and pick up the second update, then complete it.

Test case	Description
`OTAE2EImageCrashed`	This test checks that the device is able to reject an update when the image crashes.

PAL tests

Prerequisites

To port the Network Transport Interface tests, you need the following:

- A project that can build FreeRTOS with a valid FreeRTOS kernel port.
- A working implementation of OTA PAL.

Porting

- Add FreeRTOS-Libraries-Integration-Tests as a submodule into your project. The location of the submodule in the project must be where it can be built.
- Copy `config_template/test_execution_config_template.h` and `config_template/test_param_config_template.h` to a location in the build path, and rename them to `test_execution_config.h` and `test_param_config.h`.
- Include relevant files in the build system. If using `CMake`, `qualification_test.cmake` and `src/ota_pal_tests.cmake` can be used to include relevant files.
- Configure the test by implementing the following functions:
  - `SetupOtaPalTestParam()`: defined in `src/ota/ota_pal_test.h`. The implementation must have the same name and signature as defined in `ota_pal_test.h`. Currently, you do not need to configure this function.
- Implement **UNITY_OUTPUT_CHAR** so that test output logs do not interleave with device logs.
- Call `RunQualificationTest()` from the application. The device hardware must be properly initialized, and the network must be connected before the call.

Testing

This section describes the local testing of the OTA PAL qualification tests.

Enable the test

Open `test_execution_config.h` and define **OTA_PAL_TEST_ENABLED** to 1.

In `test_param_config.h`, update the following options:

- **OTA_PAL_TEST_CERT_TYPE**: Select the certificate type used.
- **OTA_PAL_CERTIFICATE_FILE**: Path to the device certificate, if applicable.
- **OTA_PAL_FIRMWARE_FILE**: Name of the firmware file, if applicable.
- **OTA_PAL_USE_FILE_SYSTEM**: Set to 1 if the OTA PAL uses file system abstraction.

Build and flash the application using a tool chain of your choice. When the `RunQualificationTest()` is called, the OTA PAL tests will run. The test results are output to the serial port.

Integrating OTA tasks

- Add OTA agent to your current MQTT demo.
- Run OTA End to End (E2E) tests with Amazon IoT. This verifies if the integration is working as expected.

> **Note**
>
> To officially qualify a device for FreeRTOS, you must validate the device's ported source code against OTA PAL and OTA E2E test groups with Amazon IoT Device Tester. Follow the instructions in Using Amazon IoT Device Tester for FreeRTOS in the *FreeRTOS User Guide* to set up Amazon IoT 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 Amazon IoT Device Tester `configs` folder.

IoT device bootloader

You must provide your own secure bootloader application. Make sure that the design and implementation provide proper mitigation to security threats. Below is the threat modeling for your reference.

Threat modeling for the IoT device bootloader

Background

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

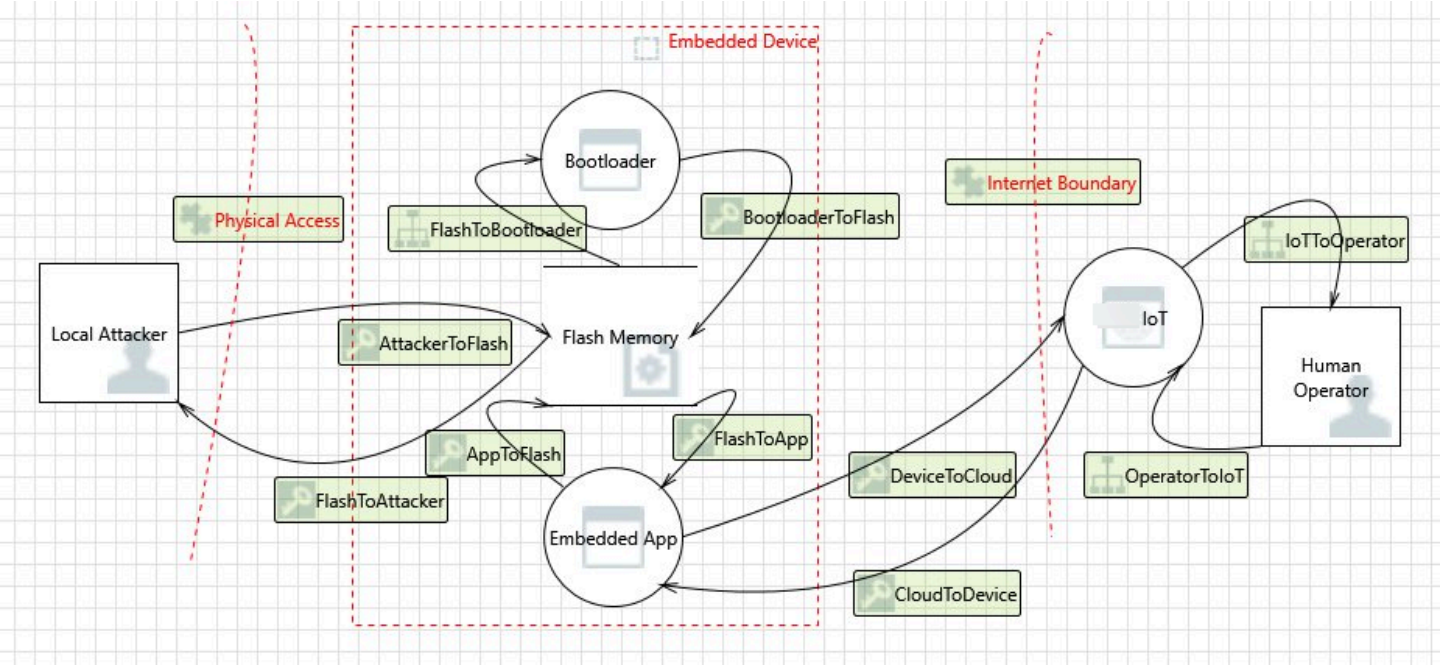
Threat modeling is an approach to security from the point of view of a hypothetical adversary. By considering the adversary's goals and methods, a threat list is created. Threats are attacks against a resource or asset performed by an adversary. The list is prioritized and used to identify and create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

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

Bootloader use cases

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

Data Flow Diagram



Threats

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

Threats:

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

  #### Mitigation example

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

  #### Mitigation examples

  - Upon boot, the bootloader verifies, as previously described. When verification fails with no previous image available, the bootloader halts.

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

  Mitigation examples

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

  Mitigation example

  - The bootloader starts a hardware watchdog timer that triggers rollback to the previous image.
- An attacker patches the bootloader to bypass image verification so the device will accept unsigned images.

  Mitigation examples

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

  Mitigation examples

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

Further threat modeling

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

Adversary's goals

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

Key assets

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

Entry points

- DHCP response
- DNS response
- MQTT over TLS
- HTTPS response
- OTA software image
- Other, as dictated by application, for example, USB
- Physical access to bus

- Decapped IC

Porting the Cellular Interface library

FreeRTOS supports the AT commands of a TCP offloaded cellular abstraction layer. For more information, see the Cellular Interface Library and Porting the Cellular Interface Library on freertos.org.

Prerequisites

There is no direct dependency for the Cellular Interface library. However, in the FreeRTOS network stack, Ethernet, Wi-Fi and cellular cannot co-exist, so developers must choose one of them to integrate with the Porting the Network Transport Interface.

> **Note**
>
> If the cellular module is able to support TLS offload, or does not support AT commands, developers can implement their own cellular abstraction to integrate with the Porting the Network Transport Interface.

Migrating from MQTT Version 3 to coreMQTT

This migration guide explains how to migrate applications from MQTT to coreMQTT.

Migrating from version 1 to version 3 for OTA applications

This guide will help you migrate your application from OTA library version 1 to version 3.

> **Note**
>
> The OTA version 2 APIs are the same as OTA v3 APIs, so if your application is using version 2 of the APIs then changes are not required for API calls but we recommend that you integrate version 3 of the library.

Demos for OTA version 3 are available here:

- ota_demo_core_mqtt.
- ota_demo_core_http.
- ota_ble.

Summary of API changes

Summary of API changes between OTA Library version 1 and version 3

OTA version 1 API	OTA version 3 API	Description of changes
OTA_AgentInit	OTA_Init	The input paramerts are changed as well as the value returned from the function due to changes in the implementation in OTA v3. Please refer to the section for OTA_Init below for details.
OTA_AgentShutdown	OTA_Shutdown	Change in the input parameters including an additional parameter for an optional unsubscribe from

OTA version 1 API	OTA version 3 API	Description of changes
		MQTT topics. Please refer to the section for OTA_Shutdown below for details.
OTA_GetAgentState	OTA_GetState	The API name is changed with no changes to the input parameter. The return value is the same but the enum and members are renamed. Please refer to the section for OTA_GetState below for details.
n/a	OTA_GetStatistics	New API added that replaces the APIs OTA_GetPacketsReceived, OTA_GetPacketsQueued, OTA_GetPacketsProcessed, OTA_GetPacketsDropped. Please refer to the section for OTA_GetStatistics below for details.
OTA_GetPacketsReceived	n/a	This API is removed from version 3 and replaced by OTA_GetStatistics.
OTA_GetPacketsQueued	n/a	This API is removed from version 3 and replaced by OTA_GetStatistics.
OTA_GetPacketsProcessed	n/a	This API is removed from version 3 and replaced by OTA_GetStatistics.

OTA version 1 API	OTA version 3 API	Description of changes
OTA_GetPacketsDropped	n/a	This API is removed from version 3 and replaced by OTA_GetStatistics.
OTA_ActivateNewImage	OTA_ActivateNewImage	The input parameters are the same but the return OTA error code is renamed and new error codes are added in version 3 of the OTA library. Please see the section for OTA_ActivateNewImage for details.
OTA_SetImageState	OTA_SetImageState	The input parameters are the same and renamed, the return OTA error code is renamed and new error codes are added in version 3 of the OTA library. Please see the section for OTA_SetImageState for details.
OTA_GetImageState	OTA_GetImageState	The input parameters are the same, the return enum is renamed in version 3 of the OTA library. Please see the section for OTA_GetImageState for details.

OTA version 1 API	OTA version 3 API	Description of changes
OTA_Suspend	OTA_Suspend	The input parameters are the same, the return OTA error code is renamed and new error codes are added in version 3 of the OTA library. Please see the section for OTA_Suspend for details.
OTA_Resume	OTA_Resume	The input parameter for connection is removed as the connection is handled in the OTA demo/application, the return OTA error code is renamed and new error codes are added in version 3 of the OTA library. Please see the section for OTA_Resume for details.
OTA_CheckForUpdate	OTA_CheckForUpdate	The input parameters are the same, the return OTA error code is renamed and new error codes are added in version 3 of the OTA library. Please see the section for OTA_CheckForUpdate for details.
n/a	OTA_EventProcessingTask	New API added and it is the main event loop to handle events for OTA update and must be called by the application task. Please see the section for OTA_Event ProcessingTask for details.

OTA version 1 API	OTA version 3 API	Description of changes
n/a	OTA_SignalEvent	New API added and it adds the event to the back of OTA event queue and is used by internal OTA modules to signal the agent task. Please see the section for OTA_SignalEvent for details.
n/a	OTA_Err_strerror	New API for error code to string conversion for OTA errors.
n/a	OTA_JobParse_strerror	New API for error code to string conversion for Job Parsing errors.
n/a	OTA_OsStatus_strerror	New API for status code to string conversion for OTA OS port status.
n/a	OTA_PalStatus_strerror	New API for status code to string conversion for OTA PAL port status.

Description of changes required

OTA_Init

When initializing the OTA Agent in v1 the `OTA_AgentInit` API is used which takes parameters for connection context, thing name, complete callback and timeout as input.

```
OTA_State_t OTA_AgentInit( void * pvConnectionContext,
                           const uint8_t * pucThingName,
                           pxOTACompleteCallback_t xFunc,
                           TickType_t xTicksToWait );
```

This API is now changed to `OTA_Init` with parameters for the buffers required for ota, ota interfaces, thing name and application callback.

```
OtaErr_t OTA_Init( OtaAppBuffer_t * pOtaBuffer,
                   OtaInterfaces_t * pOtaInterfaces,
                   const uint8_t * pThingName,
                   OtaAppCallback OtaAppCallback );
```

Removed input parameters -

pvConnectionContext -

The connection context is removed because the OTA Library Version 3 does not require the connection context to be passed to it and the MQTT/HTTP operations are handled by their respective interfaces in the OTA demo/application.

xTicksToWait -

The ticks to wait parameter is also removed as the task is created in the OTA demo/application before calling OTA_Init.

Renamed input parameters -

xFunc -

The parameter is renamed to OtaAppCallback and its type is changed to OtaAppCallback_t.

New input parameters -

pOtaBuffer

The application must allocate the buffers and pass them to the OTA library using the OtaAppBuffer_t structure during initialization. The buffers required differ slightly depending on the protocol used for downloading the file. For the MQTT protocol the buffers for stream name are required and for the HTTP protocol the buffers for pre-signed url and authorization scheme are required.

Buffers required when using MQTT for file download -

```
static OtaAppBuffer_t otaBuffer =
{
    .pUpdateFilePath    = updateFilePath,
    .updateFilePathsize = otaexampleMAX_FILE_PATH_SIZE,
    .pCertFilePath      = certFilePath,
    .certFilePathSize   = otaexampleMAX_FILE_PATH_SIZE,
```

```
    .pStreamName        = streamName,
    .streamNameSize     = otaexampleMAX_STREAM_NAME_SIZE,
    .pDecodeMemory      = decodeMem,
    .decodeMemorySize   = ( 1U << otaconfigLOG2_FILE_BLOCK_SIZE ),
    .pFileBitmap        = bitmap,
    .fileBitmapSize     = OTA_MAX_BLOCK_BITMAP_SIZE
};
```

Buffers required when using HTTP for file download -

```
static OtaAppBuffer_t otaBuffer =
{
    .pUpdateFilePath    = updateFilePath,
    .updateFilePathsize = otaexampleMAX_FILE_PATH_SIZE,
    .pCertFilePath      = certFilePath,
    .certFilePathSize   = otaexampleMAX_FILE_PATH_SIZE,
    .pDecodeMemory      = decodeMem,
    .decodeMemorySize   = ( 1U << otaconfigLOG2_FILE_BLOCK_SIZE ),
    .pFileBitmap        = bitmap,
    .fileBitmapSize     = OTA_MAX_BLOCK_BITMAP_SIZE,
    .pUrl               = updateUrl,
    .urlSize            = OTA_MAX_URL_SIZE,
    .pAuthScheme        = authScheme,
    .authSchemeSize     = OTA_MAX_AUTH_SCHEME_SIZE
};
```

Where -

```
    pUpdateFilePath    Path to store the files.
    updateFilePathsize Maximum size of the file path.
    pCertFilePath      Path to certificate file.
    certFilePathSize   Maximum size of the certificate file path.
    pStreamName        Name of stream to download the files.
    streamNameSize     Maximum size of the stream name.
    pDecodeMemory      Place to store the decoded files.
    decodeMemorySize   Maximum size of the decoded files buffer.
    pFileBitmap        Bitmap of the parameters received.
    fileBitmapSize     Maximum size of the bitmap.
    pUrl               Presigned url to download files from S3.
    urlSize            Maximum size of the URL.
    pAuthScheme        Authentication scheme used to validate download.
    authSchemeSize     Maximum size of the auth scheme.
```

pOtaInterfaces

The second input parameter to OTA_Init is a reference to the OTA interfaces for type OtaInterfaces_t. This set of interfaces must be passed to the OTA Library and includes in the operating system interface the MQTT interface, HTTP interface and platform abstraction layer interface.

OTA OS Interface

The OTA OS Functional interface is a set of APIs that must be implemented for the device to use the OTA library. The function implementations for this interface are provided to the OTA library in the user application. The OTA library calls the function implementations to perform functionalities that are typically provided by an operating system. This includes managing events, timers, and memory allocation. The implementations for FreeRTOS and POSIX are provided with the OTA library.

Example for FreeRTOS using the provided FreeRTOS port -

```
OtaInterfaces_t otaInterfaces;
otaInterfaces.os.event.init   = OtaInitEvent_FreeRTOS;
otaInterfaces.os.event.send   = OtaSendEvent_FreeRTOS;
otaInterfaces.os.event.recv   = OtaReceiveEvent_FreeRTOS;
otaInterfaces.os.event.deinit = OtaDeinitEvent_FreeRTOS;
otaInterfaces.os.timer.start  = OtaStartTimer_FreeRTOS;
otaInterfaces.os.timer.stop   = OtaStopTimer_FreeRTOS;
otaInterfaces.os.timer.delete = OtaDeleteTimer_FreeRTOS;
otaInterfaces.os.mem.malloc   = Malloc_FreeRTOS;
otaInterfaces.os.mem.free     = Free_FreeRTOS;
```

Example for Linux using the provided POSIX port -

```
OtaInterfaces_t otaInterfaces;
otaInterfaces.os.event.init    = Posix_OtaInitEvent;
otaInterfaces.os.event.send    = Posix_OtaSendEvent;
otaInterfaces.os.event.recv    = Posix_OtaReceiveEvent;
otaInterfaces.os.event.deinit  = Posix_OtaDeinitEvent;
otaInterfaces.os.timer.start   = Posix_OtaStartTimer;
otaInterfaces.os.timer.stop    = Posix_OtaStopTimer;
otaInterfaces.os.timer.delete  = Posix_OtaDeleteTimer;
otaInterfaces.os.mem.malloc    = STDC_Malloc;
otaInterfaces.os.mem.free      = STDC_Free;
```

MQTT Interface

The OTA MQTT interface is a set of APIs that must be implemented in a library to enable the OTA library to download a file block from streaming service.

Example using the coreMQTT Agent APIs from the OTA over MQTT demo-

```
OtaInterfaces_t otaInterfaces;
otaInterfaces.mqtt.subscribe = prvMqttSubscribe;
otaInterfaces.mqtt.publish = prvMqttPublish;
otaInterfaces.mqtt.unsubscribe = prvMqttUnSubscribe;
```

HTTP Interface

The OTA HTTP interface is a set of APIs that must be implemented in a library to enable the OTA library to download a file block by connecting to a pre-signed url and fetching data blocks. It is optional unless you configure the OTA library to download from a pre-signed URL instead of a streaming service.

Example using the coreHTTP APIs from the OTA over HTTP demo-

```
OtaInterfaces_t otaInterfaces;
otaInterfaces.http.init = httpInit;
otaInterfaces.http.request = httpRequest;
otaInterfaces.http.deinit = httpDeinit;
```

OTA PAL Interface

The OTA PAL interface is a set of APIs that must be implemented for the device to use the OTA library. The device specific implementation for the OTA PAL is provided to the library in the user application. These functions are used by the library to store, manage, and authenticate downloads.

```
  OtaInterfaces_t otaInterfaces;
  otaInterfaces.pal.getPlatformImageState = otaPal_GetPlatformImageState;
  otaInterfaces.pal.setPlatformImageState = otaPal_SetPlatformImageState;
  otaInterfaces.pal.writeBlock = otaPal_WriteBlock;
  otaInterfaces.pal.activate = otaPal_ActivateNewImage;
  otaInterfaces.pal.closeFile = otaPal_CloseFile;
  otaInterfaces.pal.reset = otaPal_ResetDevice;
  otaInterfaces.pal.abort = otaPal_Abort;
```

```
    otaInterfaces.pal.createFile = otaPal_CreateFileForRx;
```

Changes in return -

The return is changed from OTA agent state to OTA error code. Please refer to Amazon IoT Over-the-air Update v3.0.0 : OtaErr_t.

OTA_Shutdown

In the OTA Library version 1 the API used to shutdown the OTA Agent was OTA_AgentShutdown which is now changed to OTA_Shutdown along with changes in input parameters.

OTA Agent Shutdown (version 1)

```
OTA_State_t OTA_AgentShutdown( TickType_t xTicksToWait );
```

OTA Agent Shutdown (version 3)

```
OtaState_t OTA_Shutdown( uint32_t ticksToWait,
                         uint8_t unsubscribeFlag );
```

ticksToWait -

The number of ticks to wait for the OTA Agent to complete the shutdown process. If this is set to zero, the function will return immediately without waiting. The actual state is returned to the caller. The agent does not sleep for this while but used for busy looping.

New input parameter -

unsubscribeFlag -

Flag to indicate if unsubscribe operations should be performed from the job topics when shutdown is called. If the flag is 0 then unsubscribe operations are not called for job topics. If the application must be unsubscribed from the job topics then this flag must be set to 1 when calling OTA_Shutdown.

Changes in return -

OtaState_t -

The enum for OTA Agent state and its members are renamed. Please refer to Amazon IoT Over-the-air Update v3.0.0.

OTA_GetState

The API name is changed from OTA_AgentGetState to OTA_GetState.

OTA Agent Shutdown (version 1)

```
OTA_State_t OTA_GetAgentState( void );
```

OTA Agent Shutdown (version 3)

```
OtaState_t OTA_GetState( void );
```

Changes in return -

OtaState_t -

The enum for OTA Agent state and its members are renamed. Please refer to Amazon IoT Over-the-air Update v3.0.0.

OTA_GetStatistics

New single API added for statistics. It replaces the APIs OTA_GetPacketsReceived, OTA_GetPacketsQueued, OTA_GetPacketsProcessed, OTA_GetPacketsDropped. Also, in the OTA Library version 3, the statistics numbers are related to the current job only.

OTA Library version 1

```
uint32_t OTA_GetPacketsReceived( void );
uint32_t OTA_GetPacketsQueued( void );
uint32_t OTA_GetPacketsProcessed( void );
uint32_t OTA_GetPacketsDropped( void );
```

OTA Library version 3

```
OtaErr_t OTA_GetStatistics( OtaAgentStatistics_t * pStatistics );
```

pStatistics -

Input/output parameter for statistics data like packets received, dropped, queued and processed for current job.

Output parameter -

OTA error code.

Example usage -

```
OtaAgentStatistics_t otaStatistics = { 0 };
OTA_GetStatistics( &otaStatistics );
LogInfo( ( " Received: %u   Queued: %u   Processed: %u   Dropped: %u",
                           otaStatistics.otaPacketsReceived,
                           otaStatistics.otaPacketsQueued,
                           otaStatistics.otaPacketsProcessed,
                           otaStatistics.otaPacketsDropped ) );
```

OTA_ActivateNewImage

The input parameters are the same but the return OTA error code is renamed and new error codes are added in the version 3 of the OTA library.

OTA Library version 1

```
OTA_Err_t OTA_ActivateNewImage( void );
```

OTA Library version 3

```
OtaErr_t OTA_ActivateNewImage( void );
```

The return OTA error code enum is changed and new error codes are added. Please refer to Amazon IoT Over-the-air Update v3.0.0 : OtaErr_t.

Example usage -

```
OtaErr_t otaErr = OtaErrNone;
otaErr = OTA_ActivateNewImage();
/* Handle error */
```

OTA_SetImageState

The input parameters are the same and renamed, the return OTA error code is renamed and new error codes are added in the version 3 of the OTA library.

OTA Library version 1

```
OTA_Err_t OTA_SetImageState( OTA_ImageState_t eState );
```

OTA Library version 3

```
OtaErr_t OTA_SetImageState( OtaImageState_t state );
```

The input parameter is renamed to OtaImageState_t. Please refer to Amazon IoT Over-the-air Update v3.0.0.

The return OTA error code enum is changed and new error codes are added. Please refer to Amazon IoT Over-the-air Update v3.0.0 / OtaErr_t.

Example usage -

```
  OtaErr_t otaErr = OtaErrNone;
  otaErr = OTA_SetImageState( OtaImageStateAccepted );
  /* Handle error */
```

OTA_GetImageState

The input parameters are same, the return enum is renamed in the version 3 of the OTA library.

OTA Library version 1

```
OTA_ImageState_t OTA_GetImageState( void );
```

OTA Library version 3

```
OtaImageState_t OTA_GetImageState( void );
```

The return enum is renamed to OtaImageState_t. Please refer to Amazon IoT Over-the-air Update v3.0.0 : OtaImageState_t .

Example usage -

```
 OtaImageState_t imageState;
 imageState = OTA_GetImageState();
```

OTA_Suspend

The input parameters are the same, the return OTA error code is renamed and new error codes are added in the version 3 of the OTA library.

OTA Library version 1

```
OTA_Err_t OTA_Suspend( void );
```

OTA Library version 3

```
OtaErr_t OTA_Suspend( void );
```

The return OTA error code enum is changed and new error codes are added. Please refer to Amazon IoT Over-the-air Update v3.0.0 : OtaErr_t.

Example usage -

```
OtaErr_t xOtaError = OtaErrUninitialized;
xOtaError = OTA_Suspend();
/* Handle error */
```

OTA_Resume

The input parameter for connection is removed as the connection is handled in the OTA demo/application, the return OTA error code is renamed and new error codes are added in the version 3 of the OTA library.

OTA Library version 1

```
OTA_Err_t OTA_Resume( void * pxConnection );
```

OTA Library version 3

```
OtaErr_t OTA_Resume( void );
```

The return OTA error code enum is changed and new error codes are added. Please refer to Amazon IoT Over-the-air Update v3.0.0 : OtaErr_t.

Example usage -

```
OtaErr_t xOtaError = OtaErrUninitialized;
xOtaError = OTA_Resume();
/* Handle error */
```

OTA_CheckForUpdate

The input parameters are the same, the return OTA error code is renamed and new error codes are added in the version 3 of the OTA library.

OTA Library version 1

```
OTA_Err_t OTA_CheckForUpdate( void );
```

OTA Library version 3

```
OtaErr_t OTA_CheckForUpdate( void )
```

The return OTA error code enum is changed and new error codes are added. Please refer to Amazon IoT Over-the-air Update v3.0.0 : OtaErr_t.

OTA_EventProcessingTask

This is a new API and is the main event loop to handle events for OTA updates. It must be called by the application task. This loop will continue to handle and execute events received for OTA Update until this task is terminated by the application.

OTA Library version 3

```
void OTA_EventProcessingTask( void * pUnused );
```

Example for FreeRTOS -

```
/* Create FreeRTOS task*/
xTaskCreate( prvOTAAgentTask,
             "OTA Agent Task",
```

```
             otaexampleAGENT_TASK_STACK_SIZE,
             NULL,
             otaexampleAGENT_TASK_PRIORITY,
             NULL );

/* Call OTA_EventProcessingTask from the task */
static void prvOTAAgentTask( void * pParam )
{
    /* Calling OTA agent task. */
    OTA_EventProcessingTask( pParam );
    LogInfo( ( "OTA Agent stopped." ) );

    /* Delete the task as it is no longer required. */
    vTaskDelete( NULL );
}
```

Example for POSIX -

```
/* Create posix thread.*/
if( pthread_create( &threadHandle, NULL, otaThread, NULL ) != 0 )
{
    LogError( ( "Failed to create OTA thread: "
                ",errno=%s",
                strerror( errno ) ) );

   /* Handle error. */
}

/* Call OTA_EventProcessingTask from the thread.*/
static void * otaThread( void * pParam )
{
    /* Calling OTA agent task. */
    OTA_EventProcessingTask( pParam );
    LogInfo( ( "OTA Agent stopped." ) );

    return NULL;
}
```

OTA_SignalEvent

This is a new API that adds the event to the back of the event queue and is also used by internal OTA modules to signal agent task.

OTA Library version 3

```
bool OTA_SignalEvent( const OtaEventMsg_t * const pEventMsg );
```

Example usage -

```
OtaEventMsg_t xEventMsg = { 0 };
xEventMsg.eventId = OtaAgentEventStart;
( void ) OTA_SignalEvent( &xEventMsg );
```

Integrating the OTA Library as a submodule in your application

If you want to integrate the OTA library in your own application you can use the git submodule command. Git submodules allow you to keep a Git repository as a subdirectory of another Git repository. The OTA library version 3 is maintained in the ota-for-aws-iot-embedded-sdk repository.

```
git submodule add https://github.com/aws/ota-for-aws-iot-embedded-
sdk.git destination_folder
```

```
git commit -m "Added the OTA Library as submodule to the project."
```

```
git push
```

For more information, see Integrating the OTA Agent into your application in the *FreeRTOS User Guide*.

References

- OTAv1.
- OTAv3.

Migrating from version 1 to version 3 for OTA PAL port

The Over-the-air Updates Library introduced some changes in the folder structure and the placement of configurations required by the library and the demo applications. For OTA applications designed to work with v1.2.0 to migrate to v3.0.0 of the library, you must update the PAL port function signatures and include additional configuration files as described in this migration guide.

Changes to OTA PAL

- The OTA PAL port directory name has been updated from `ota` to `ota_pal_for_aws`. This folder must contain 2 files: `ota_pal.c` and `ota_pal.h`. The PAL header file `libraries/freertos_plus/aws/ota/src/aws_iot_ota_pal.h` has been deleted from the OTA library and must be defined inside the port.
- The return codes (`OTA_Err_t`) are translated into an enum `OTAMainStatus_t`. Refer to ota_platform_interface.h for translated return codes. Helper macros are also provided to combine `OtaPalMainStatus` and `OtaPalSubStatus` codes and extract `OtaMainStatus` from `OtaPalStatus` and similar.
- Logging in the PAL
  - Removed the `DEFINE_OTA_METHOD_NAME` macro.
  - Earlier: `OTA_LOG_L1( "[%s] Receive file created.\r\n", OTA_METHOD_NAME );`.
  - Updated: `LogInfo(( "Receive file created."));` Use `LogDebug`, `LogWarn` and `LogError` for the appropriate log.
- Variable `cOTA_JSON_FileSignatureKey` changed to `OTA_JsonFileSignatureKey`.

Functions

The function signatures are defined in `ota_pal.h` and start with the prefix `otaPal` instead of `prvPAL`.

> **Note**
>
> The exact name of the PAL is technically open ended, but to be compatible with the qualification tests, the name should conform to the ones specified below.

- Version 1: `OTA_Err_t prvPAL_CreateFileForRx( OTA_FileContext_t * const *C* );`

  Version 3: `OtaPalStatus_t otaPal_CreateFileForRx( OtaFileContext_t * const *pFileContext* );`

  Notes: Create a new receive file for the data chunks as they come in.
- Version 1: `int16_t prvPAL_WriteBlock( OTA_FileContext_t * const C, uint32_t ulOffset, uint8_t * const pcData, uint32_t ulBlockSize );`

  Version 3: `int16_t otaPal_WriteBlock( OtaFileContext_t * const pFileContext, uint32_t ulOffset, uint8_t * const pData, uint32_t ulBlockSize );`

  Notes: Write a block of data to the specified file at the given offset.
- Version 1: `OTA_Err_t prvPAL_ActivateNewImage( void );`

  Version 3: `OtaPalStatus_t otaPal_ActivateNewImage( OtaFileContext_t * const *pFileContext* );`

  Notes: Activate the newest MCU image received via OTA.
- Version 1: `OTA_Err_t prvPAL_ResetDevice( void );`

  Version 3: `OtaPalStatus_t otaPal_ResetDevice( OtaFileContext_t * const *pFileContext* );`

  Notes: Reset the device.
- Version 1: `OTA_Err_t prvPAL_CloseFile( OTA_FileContext_t * const *C* );`

  Version 3: `OtaPalStatus_t otaPal_CloseFile( OtaFileContext_t * const *pFileContext* );`

  Notes: Authenticate and close the underlying receive file in the specified OTA context.
- Version 1: `OTA_Err_t prvPAL_Abort( OTA_FileContext_t * const *C* );`

  Version 3: `OtaPalStatus_t otaPal_Abort( OtaFileContext_t * const *pFileContext* );`

  Notes: Stop an OTA transfer.

- Version 1: `OTA_Err_t prvPAL_SetPlatformImageState( OTA_ImageState_t *eState* );`

  Version 3: `OtaPalStatus_t otaPal_SetPlatformImageState( OtaFileContext_t * const pFileContext, OtaImageState_t eState );`

  Notes: Attempt to set the state of the OTA update image.
- Version 1: `OTA_PAL_ImageState_t prvPAL_GetPlatformImageState( void );`

  Version 3: `OtaPalImageState_t otaPal_GetPlatformImageState( OtaFileContext_t * const *pFileContext* );`

  Notes: Get the state of the OTA update image.

Data Types

- Version 1: `OTA_PAL_ImageState_t`

  File: `aws_iot_ota_agent.h`

  Version 3: `OtaPalImageState_t`

  File: `ota_private.h`

  Notes: *The image state set by platform implementation.*
- Version 1: `OTA_Err_t`

  File: `aws_iot_ota_agent.h`

  Version 3: `OtaErr_t OtaPalStatus_t (combination of OtaPalMainStatus_t and OtaPalSubStatus_t)`

  File: `ota.h`, `ota_platform_interface.h`

  Notes: v1: These were macros defining a 32 unsigned integer. v3: Specialized enum representing the type of error and associated with an error code.
- Version 1: `OTA_FileContext_t`

  File: `aws_iot_ota_agent.h`

Version 3: `OtaFileContext_t`

File: `ota_private.h`

Notes: v1: Contains an enum and buffers for the data. v3: Contains additional data-length variables.

- Version 1: `OTA_ImageState_t`

  File: `aws_iot_ota_agent.h`

  Version 3: `OtaImageState_t`

  File: `ota_private.h`

  Notes: *OTA Image states*

Configuration changes

The file `aws_ota_agent_config.h` was renamed to `ota_config.h` which changes the include guards from `_AWS_OTA_AGENT_CONFIG_H_` to `OTA_CONFIG_H_`.

- The file `aws_ota_codesigner_certificate.h` has been deleted.
- Included the new logging stack to print debug messages:

```
/**************************************************/
/******* DO NOT CHANGE the following order ********/
/**************************************************/

/* Logging related header files are required to be included in the following order:
 * 1. Include the header file "logging_levels.h".
 * 2. Define LIBRARY_LOG_NAME and  LIBRARY_LOG_LEVEL.
 * 3. Include the header file "logging_stack.h".
 */

/* Include header that defines log levels. */
#include "logging_levels.h"

/* Configure name and log level for the OTA library. */
#ifndef LIBRARY_LOG_NAME
    #define LIBRARY_LOG_NAME     "OTA"
```

```
#endif
#ifndef LIBRARY_LOG_LEVEL
    #define LIBRARY_LOG_LEVEL    LOG_INFO
#endif

#include "logging_stack.h"

/************ End of logging configuration ****************/
```

- Added the constant config:

```
/** * @brief Size of the file data block message (excluding the header). */
#define otaconfigFILE_BLOCK_SIZE ( 1UL << otaconfigLOG2_FILE_BLOCK_SIZE )
```

New File: `ota_demo_config.h` contains the configs that are required by the OTA demo such as the code signing certificate and application version.

- `signingcredentialSIGNING_CERTIFICATE_PEM` which was defined in `demos/include/aws_ota_codesigner_certificate.h` has been moved to `ota_demo_config.h` as `otapalconfigCODE_SIGNING_CERTIFICATE` and can can be accessed from the PAL files as:

  ```
  static const char codeSigningCertificatePEM[] = otapalconfigCODE_SIGNING_CERTIFICATE;
  ```

  The file `aws_ota_codesigner_certificate.h` has been deleted.
- The macros `APP_VERSION_BUILD`, `APP_VERSION_MINOR`, `APP_VERSION_MAJOR` have been added to `ota_demo_config.h`. The old files containing the version information have been removed, for example `tests/include/aws_application_version.h`, `libraries/c_sdk/standard/common/include/iot_appversion32.h`, `demos/demo_runner/aws_demo_version.c`.

Changes to the OTA PAL tests

- Removed the "Full_OTA_AGENT" test group along with all related files. This test group was previously required for qualification. These tests were for the OTA library and not specific to the OTA PAL port. The OTA library now has full test coverage that is hosted in the OTA repository so this test group is no longer required.

- Removed the "Full_OTA_CBOR" and "Quarantine_OTA_CBOR" test groups as well as all related files. These tests were not part of the qualification tests. The functionalities these tests covered are now being tested in the OTA repository.
- Moved the testing files from the library directory to the `tests/integration_tests/ota_pal` directory.
- Updated the OTA PAL qualification tests to use v3.0.0 of the OTA library API.
- Updated how the OTA PAL tests access the code signing certificate for tests. Previously there was a dedicated header file for the code signing credential. This is no longer the case for the new version of the library. The test code expects this variable to be defined in `ota_pal.c`. The value is assigned to a macro that is defined in the platform specific OTA config file.

Checklist

Use this checklist to make sure you follow the steps required for migration:

- Update the name of the ota pal port folder from `ota` to `ota_pal_for_aws`.
- Add the file `ota_pal.h` with the functions mentioned above. For an example `ota_pal.h` file, see GitHub.
- Add the configuration files:
  - Change the name of the file from `aws_ota_agent_config.h` to (or create) `ota_config.h`.
    - Add:

      ```
      otaconfigFILE_BLOCK_SIZE ( 1UL << otaconfigLOG2_FILE_BLOCK_SIZE )
      ```

    - Include:

      ```
      #include "ota_demo_config.h"
      ```

  - Copy the above files to the `aws_test config` folder and substitute any includes of `ota_demo_config.h` with `aws_test_ota_config.h`.
  - Add an `ota_demo_config.h` file.
  - Add an `aws_test_ota_config.h` file.
- Make the following changes to `ota_pal.c`:
  - Update the includes with the latest OTA library file names.
  - Remove the `DEFINE_OTA_METHOD_NAME` macro.

  - Update the signatures of the OTA PAL functions.
  - Update the name of the file context variable from `C` to `pFileContext`.
  - Update the `OTA_FileContext_t` struct and all related variables.
  - Update `cOTA_JSON_FileSignatureKey` to `OTA_JsonFileSignatureKey`.
  - Update the `OTA_PAL_ImageState_t` and `Ota_ImageState_t` types.
  - Update the error type and values.
  - Update the printing macros to use the logging stack.
  - Update the `signingcredentialSIGNING_CERTIFICATE_PEM` to be `otapalconfigCODE_SIGNING_CERTIFICATE`.
  - Update `otaPal_CheckFileSignature` and `otaPal_ReadAndAssumeCertificate` function comments.
- Update the `CMakeLists.txt` file.
- Update the IDE projects.

Document history

The following table describes the documentation history for the FreeRTOS Porting Guide and the FreeRTOS Qualification Guide.

Date	Documentation version	Change history	FreeRTOS version
May, 2022	*FreeRTOS Porting Guide* *FreeRTOS Qualification Guide*	• Updated existing tests, added new tests, and removed redundant tests based on FreeRTOS Long Term Support (LTS) libraries. For more information, see FreeRTOS Libraries Integration Tests 202205.00 on GitHub. • Updated FreeRTOS porting flowchart. • Added a new Porting the Network Transport Interface. • Porting the Amazon IoT over-the-air (OTA) update library is now required for qualification. • Removed Wi-Fi, and TLS abstraction porting guide	202012.04-LTS 202112.00

Date	Documentation version	Change history	FreeRTOS version
		as it is not required any more. • See Latest changes for further updates on FreeRTOS qualification.	
July, 2021	202107.00 (Porting Guide) 202107.00 (Qualification Guide)	• Release 202107.00 • Changed Porting the Amazon IoT over-the-air (OTA) update library • Added Migrating from version 1 to version 3 for OTA applications • Added Migrating from version 1 to version 3 for OTA PAL port	202107.00
December, 2020	202012.00 (Porting Guide) 202012.00 (Qualification Guide)	• Release 202012.00 • Added Configuring the coreHTTP library • Added Porting the Cellular Interface library	202012.00

Date	Documentation version	Change history	FreeRTOS version
November, 2020	202011.00 (Porting Guide) 202011.00 (Qualification Guide)	• Release 202011.00 • Added Configuring the coreMQTT library	202011.00
July, 2020	202007.00 (Porting Guide) 202007.00 (Qualification Guide)	• Release 202007.00	202007.00
February 18, 2020	202002.00 (Porting Guide) 202002.00 (Qualification Guide)	• Release 202002.00 • Amazon FreeRTOS is now FreeRTOS	202002.00
December 17, 2019	201912.00 (Porting Guide) 201912.00 (Qualification Guide)	• Release 201912.00 • Added Porting of the common I/O libraries.	201912.00
October 29, 2019	201910.00 (Porting Guide) 201910.00 (Qualification Guide)	• Release 201910.00 • Updated random number generator porting information.	201910.00

Date	Documentation version	Change history	FreeRTOS version
August 26, 2019	201908.00 (Porting Guide) 201908.00 (Qualification Guide)	• Release 201908.00 • Added *Configuring the HTTPS client library for testing* Updated Porting the corePKCS11 library	201908.00
June 17, 2019	201906.00 (Porting Guide) 201906.00 (Qualification Guide)	• Release 201906.00 • Directory structured updated	201906.00 Major
May 21, 2019	1.4.8 (Porting Guide) 1.4.8 (Qualification Guide)	• Porting documentation moved to the FreeRTOS Porting Guide • Qualification documentation moved to the FreeRTOS Qualification Guide	1.4.8
February 25, 2019	1.1.6	• Removed download and configuration instructions from Getting Started Guide Template Appendix (page 84)	1.4.5 1.4.6 1.4.7

Date	Documentation version	Change history	FreeRTOS version
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 Bluetooth Low Energy porting appendix (page 52) • Added Amazon IoT Device Tester for FreeRTOS testing informati on throughout document • Added CMake link to Information for listing on the FreeRTOS Console appendix (page 85)	1.4.4

Date	Documentation version	Change history	FreeRTOS version
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	Documentation version	Change history	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

Date	Documentation version	Change history	FreeRTOS version
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
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 Amazon CLI config instructions to examples in Instructions to Create a BYOC appendix (page 57)	1.3.1 1.3.2

Date	Documentation version	Change history	FreeRTOS version
July 31, 2018	1.0.0	Initial version of the FreeRTOS Qualification Program Guide	1.3.0