aws

User Guide

FreeRTOS



Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

FreeRTOS: User Guide

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

What is FreeRTOS? 1
Download FreeRTOS source code 1
FreeRTOS-qualified hardware platforms 1
Additional resources 2
FreeRTOS versions 3
FreeRTOS Long Term Support 3
FreeRTOS Extended Maintenance Plan 3
FreeRTOS architecture 4
Development workflow 4
FreeRTOS kernel fundamentals 6
The FreeRTOS kernel scheduler 6
Kernel memory allocation 7
Manage application memory 7
Intertask coordination 8
Queues 8
Semaphores and mutexes 9
Direct-to-task notifications 9
Stream buffers 10
Message buffers 11
Symmetric multiprocessing (SMP) support 13
Software timers 13
Low power support 14
`FreeRTOSConfig.h` 14
AWS IoT Device SDK for Embedded C 15
Understand the FreeRTOS Common IO APIs 16
Libraries 16
Common IO - basic 16
Common IO - BLE 18
Common IO for Amazon Common Software 18
What is ACS? 18
Qualification Program 19
Get started with FreeRTOS 20
Get started with Quick Connect 20
Explore FreeRTOS libraries 20

Build a secure and robust AWS IoT product ... 20
Develop your AWS IoT application ... 21
AWS IoT Device Tester for FreeRTOS ... 22
FreeRTOS qualification suite ... 22
Understand custom test suites ... 23
Supported versions of IDT for FreeRTOS ... 24
Latest version of IDT for FreeRTOS ... 24
Earlier IDT versions ... 26
Unsupported IDT versions ... 32
Download IDT for FreeRTOS ... 60
Download IDT manually ... 61
Download IDT programmatically ... 61
IDT with FreeRTOS qualification suite 2.0 (FRQ 2.0) ... 67
Set up the LTS qualification prerequisites ... 68
First test of your microcontroller board ... 75
Use the IDT UI to run the FreeRTOS qualification suite ... 92
Run the FreeRTOS qualification 2.0 suite ... 106
View the IDT for FreeRTOSresults ... 109
Interpret the IDT for FreeRTOS results ... 110
View the IDT for FreeRTOSlogs ... 113
IDT with FreeRTOS qualification suite 1.0 (FRQ 1.0) ... 113
Setup the 1.0 qualification prerequisites ... 115
First test of your microcontroller board ... 119
Use the IDT UI to run the FreeRTOS qualification suite ... 139
Run Bluetooth Low Energy tests ... 149
Run the FreeRTOS qualification suite ... 154
View the IDT for FreeRTOS results ... 160
Interpret the IDT for FreeRTOS results ... 161
View the IDT for FreeRTOS logs ... 164
Develop and run your own IDT test suites ... 164
Download the latest version of IDT for FreeRTOS ... 165
Test suite workflow ... 165
Tutorial: Build and run the sample IDT test suite ... 166
Tutorial: Develop a simple IDT test suite ... 171
Test suite versions ... 256
Troubleshoot errors ... 257

Troubleshoot device configuration ... 258
Troubleshoot timeout errors ... 271
Cellular feature and AWS charges ... 272
Qualification report generation policy ... 272
AWS Managed policy for AWS IoT Device Tester ... 272
Managed policy ... 273
Policy updates ... 280
Support policy ... 282
Security in AWS ... 283
Identity and Access Management ... 283
Audience ... 284
Authenticating with identities ... 285
Managing access using policies ... 288
How FreeRTOS works with IAM ... 290
Identity-based policy examples ... 297
Troubleshooting ... 300
Compliance validation ... 301
Resilience ... 302
Infrastructure security ... 303
Amazon-FreeRTOS Github Repository Migration Guide ... 304
Appendix ... 304
Archive ... 310
FreeRTOS User Guide Archive ... 310
Previous FreeRTOS User Guide contents ... 310
Get Started with FreeRTOS ... 310
Over-the-Air Updates ... 505
FreeRTOS Libraries ... 587
FreeRTOS demos ... 653

What is FreeRTOS?

Developed in partnership with the world's leading chip companies over a 15-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. FreeRTOS also includes demo applications that show FreeRTOS features on qualified boards.

FreeRTOS is an open-source project. You can download the source code, contribute changes or enhancements, or report issues on the GitHub site at https://github.com/FreeRTOS/FreeRTOS.

We release FreeRTOS code under the MIT open source license, so you can use it in commercial and personal projects.

We also welcome contributions to the FreeRTOS documentation (*FreeRTOS User Guide*, *FreeRTOS Porting Guide*, and *FreeRTOS Qualification Guide*). To view the markdown source for the documentation, see https://github.com/awsdocs/aws-freertos-docs. It's released under the Creative Commons (CC BY-ND) license.

Download FreeRTOS source code

Download the latest FreeRTOS and Long Term Support (LTS) packages from the Downloads page on freertos.org.

FreeRTOS-qualified hardware platforms

The following hardware platforms are qualified for FreeRTOS:

- ATECC608A Zero Touch Provisioning Kit for AWS IoT
- Cypress CYW943907AEVAL1F Development Kit
- Cypress CYW954907AEVAL1F Development Kit
- Cypress CY8CKIT-064S0S2-4343W Kit
- Espressif ESP32-DevKitC

- Espressif ESP-WROVER-KIT
- Espressif ESP-WROOM-32SE
- Espressif ESP32-S2-Saola-1
- Infineon XMC4800 IoT Connectivity Kit
- Marvell MW320 AWS IoT Starter Kit
- Marvell MW322 AWS IoT Starter Kit
- MediaTek MT7697Hx Development Kit
- Microchip Curiosity PIC32MZEF Bundle
- Nordic nRF52840-DK
- NuMaker-IoT-M487
- NXP LPC54018 IoT Module
- OPTIGA Trust X Security Solution
- Renesas RX65N RSK IoT Module
- STMicroelectronicsSTM32L4 Discovery Kit IoT Node
- Texas Instruments CC3220SF-LAUNCHXL
- Microsoft Windows 7 or later, with at least a dual core and a hard-wired Ethernet connection
- Xilinx Avnet MicroZed Industrial IoT Kit

Qualified devices are also listed on the AWS Partner Device Catalog.

For information about qualifying a new device, see the FreeRTOS Qualification Guide.

Additional resources

These resources might be helpful to you.

- For additional FreeRTOS Documentation, see freertos.org.
- For questions about FreeRTOS for the FreeRTOS engineering team, you can open an issue on the FreeRTOS GitHub page.
- For technical questions about FreeRTOS, see the FreeRTOS Community Forums.
- For more information about connecting devices to AWS IoT, see Device Provisioning in the AWS IoT Core Developer Guide.
- For technical support for AWS, see AWS Support.

- For questions about AWS billing, account services, events, abuse, or other issues with AWS, see the Contact Us page.

FreeRTOS versions

Individual libraries use x.y.z style version numbers, similar to semantic versioning. x is the major version number, y the minor version number, and starting from 2022, z is a patch number. Before 2022, z was a point release number, which required the first LTS libraries to have a patch number of the form *"x.y.z LTS Patch 2"*.

Library packages use yyyymm.x style date stamp version numbers. yyyy is the year, mm the month, and x an optional sequence number showing the release order within the month. In the case of the LTS package, x is a sequential patch number for that LTS release. The individual libraries contained in a package are whatever the latest version of that library was on that date. For the LTS package, it's the latest patch version of the LTS libraries originally released as an LTS version on that date.

FreeRTOS Long Term Support

FreeRTOS Long Term Support (LTS) releases receive security and critical bug fixes (should any be necessary) for at least two years following their release. With this ongoing maintenance, you can incorporate bug fixes throughout a development and deployment cycle without the expensive disruption of updating to new major versions of FreeRTOS libraries.

With FreeRTOS LTS, you get the complete set of libraries needed to build secure connected IoT and embedded products. LTS helps reduce maintenance and testing costs associated with updating libraries on your devices already in production.

FreeRTOS LTS includes the FreeRTOS kernel and IoT libraries: FreeRTOS+TCP, coreMQTT, coreHTTP, corePKCS11, coreJSON, AWS IoT OTA, AWS IoT Jobs, AWS IoT Device Defender, and AWS IoT Device Shadow. For more information, see the FreeRTOS LTS libraries.

FreeRTOS Extended Maintenance Plan

AWS also offers FreeRTOS Extended Maintenance Plan (EMP), which provides security patches and critical bug fixes on your chosen FreeRTOS Long Term Support (LTS) version for up to ten additional years. With FreeRTOS EMP, your FreeRTOS based long-lived devices can rely on a version that has feature stability and receives security updates for years. You receive timely notifications of upcoming patches on FreeRTOS libraries, so you can plan the deployment of security patches on your Internet of Things (IoT) devices.

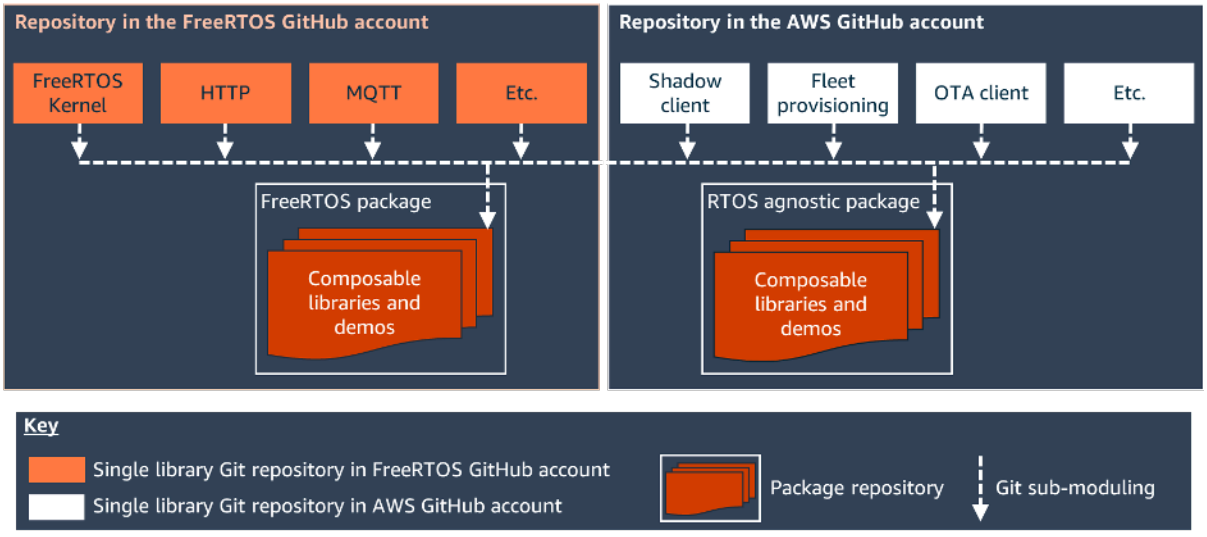
To learn more about FreeRTOS EMP, see the Features page.

FreeRTOS architecture

FreeRTOS contains two types of repositories, single library repositories and package repositories. Each single library repository contains the source code for one library without any build projects or examples. Package repositories contain multiple libraries, and can contain preconfigured projects that demonstrate the library's use.

While package repositories contain multiple libraries, they don't contain copies of those libraries. Instead, package repositories reference the libraries they contain as git submodules. Using submodules ensures that there is a single source of truth for each individual library.

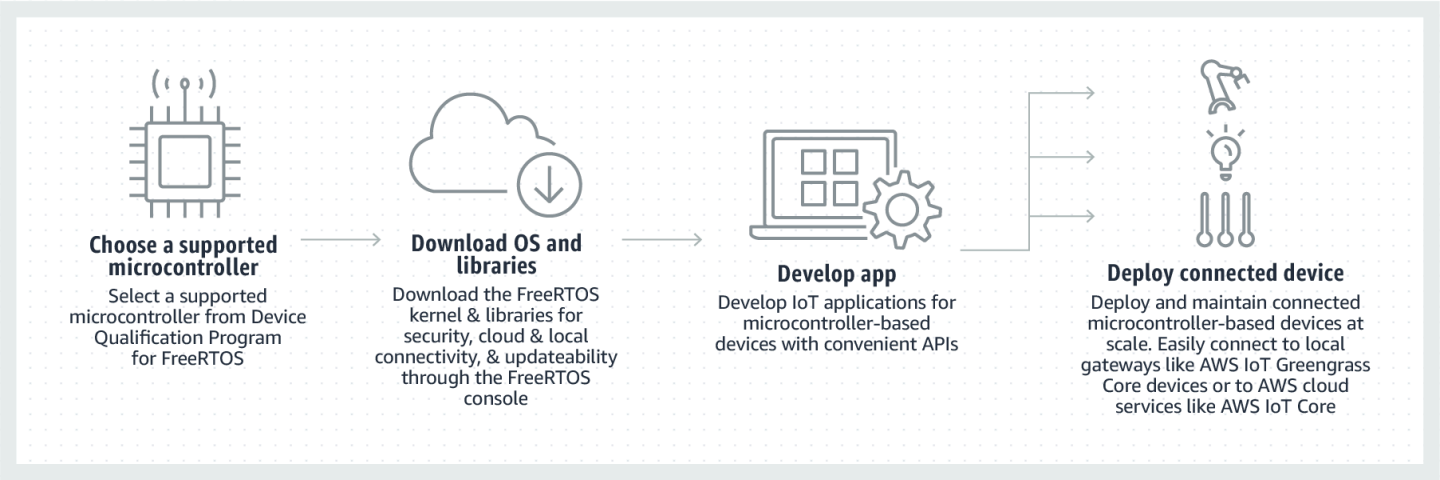
The individual library git repositories are split between two GitHub organizations. Repositories containing FreeRTOS specific libraries (such as FreeRTOS+TCP) or generic libraries (such as coreMQTT, which is cloud agnostic because it works with any MQTT broker) are in the FreeRTOS GitHub organization. Repositories containing AWS IoT specific libraries (such as the AWS IoT over-the-air update client) are in the AWS GitHub organization. The following diagram explains the structure.



Development workflow

You start development by downloading FreeRTOS. You unzip the package and import it into your IDE. You can then develop an application on your selected hardware platform and manufacture and deploy these devices using the development process appropriate for your device. Deployed

devices can connect to the AWS IoT service or AWS IoT Greengrass as part of a complete IoT solution.



FreeRTOS kernel fundamentals

The FreeRTOS kernel is a real-time operating system that supports numerous architectures. The fundamentals of it are ideal for building embedded microcontroller applications. It provides:

- A multitasking scheduler.
- Multiple memory allocation options (including the ability to create completely statically-allocated systems).
- Intertask coordination primitives, including task notifications, message queues, multiple types of semaphore, and stream and message buffers.
- Support for symmetric multiprocessing (SMP) on multi-core microcontrollers.

The FreeRTOS kernel never performs non-deterministic operations, such as walking a linked list, inside a critical section or interrupt. The FreeRTOS kernel includes an efficient software timer implementation that does not use any CPU time unless a timer needs servicing. Blocked tasks do not require time-consuming periodic servicing. Direct-to-task notifications allow fast task signaling, with practically no RAM overhead. They can be used in most inter task and interrupt-to-task signaling scenarios.

The FreeRTOS kernel is designed to be small, simple, and easy to use. A typical RTOS kernel binary image is in the range of 4000 to 9000 bytes.

For the most up-to-date documentation about the FreeRTOS kernel, see FreeRTOS.org. FreeRTOS.org offers a number of detailed tutorials and guides about using the FreeRTOS kernel, including a FreeRTOS FreeRTOS Kernel Quick Start Guide and the more in-depth RTOS Implementation in the *FreeRTOS Documentation*.

The FreeRTOS kernel scheduler

An embedded application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context, with no dependency on other tasks. Only one task in the application is running at any point in time. The real-time RTOS scheduler determines when each task should run. Each task is provided with its own stack. When a task is swapped out so another task can run, the task's execution context is saved to the task stack so it can be restored when the same task is later swapped back in to resume its execution.

To provide deterministic real-time behavior, the FreeRTOS tasks scheduler allows tasks to be assigned strict priorities. RTOS ensures the highest priority task that is able to execute is given processing time. This requires sharing processing time between tasks of equal priority if they are ready to run simultaneously. FreeRTOS also creates an idle task that executes only when no other tasks are ready to run.

Kernel memory allocation

The RTOS kernel needs RAM each time a task, queue, or other RTOS object is created. The RAM can be allocated:

- Statically at compile time.
- Dynamically from the RTOS heap by the RTOS API object creation functions.

When RTOS objects are created dynamically, using the standard C library `malloc()` and `free()` functions is not always appropriate for a number of reasons:

- They might not be available on embedded systems.
- They take up valuable code space.
- They are not typically thread-safe.
- They are not deterministic.

For these reasons, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, so you can provide an application-specific implementation appropriate for the real-time system you're developing. When the RTOS kernel requires RAM, it calls `pvPortMalloc()` instead of `malloc()`(). When RAM is being freed, the RTOS kernel calls `vPortFree()` instead of `free()`.

Manage application memory

When applications need memory, they can allocate it from the FreeRTOS heap. FreeRTOS offers several heap management schemes that range in complexity and features. You can also provide your own heap implementation.

The FreeRTOS kernel includes five heap implementations:

heap_1

Is the simplest implementation. Does not permit memory to be freed.

heap_2

Permits memory to be freed, but not does coalesce adjacent free blocks.

heap_3

Wraps the standard `malloc()` and `free()` for thread safety.

heap_4

Coalesces adjacent free blocks to avoid fragmentation. Includes an absolute address placement option.

heap_5

Is similar to heap_4. Can span the heap across multiple, non-adjacent memory areas.

Intertask coordination

This section contains information about FreeRTOS primitives.

Topics

- Queues
- Semaphores and mutexes
- Direct-to-task notifications
- Stream buffers
- Message buffers

Queues

Queues are the primary form of intertask communication. They can be used to send messages between tasks and between interrupts and tasks. In most cases, they are used as thread-safe, First In First Out (FIFO) buffers with new data being sent to the back of the queue. (Data can also be sent to the front of the queue.) Messages are sent through queues by copy, meaning the data (which can be a pointer to larger buffers) is itself copied into the queue rather than simply storing a reference to the data.

Queue APIs permit a block time to be specified. When a task attempts to read from an empty queue, the task is placed into the Blocked state until data becomes available on the queue or the block time elapses. Tasks in the Blocked state do not consume any CPU time, allowing other tasks to run. Similarly, when a task attempts to write to a full queue, the task is placed into the Blocked state until space becomes available in the queue or the block time elapses. If more than one task blocks on the same queue, the task with the highest priority is unblocked first.

Other FreeRTOS primitives, such as direct-to-task notifications and stream and message buffers, offer lightweight alternatives to queues in many common design scenarios.

Semaphores and mutexes

The FreeRTOS kernel provides binary semaphores, counting semaphores, and mutexes for both mutual exclusion and synchronization purposes.

Binary semaphores can only have two values. They are a good choice for implementing synchronization (either between tasks or between tasks and an interrupt). Counting semaphores take more than two values. They allow many tasks to share resources or perform more complex synchronization operations.

Mutexes are binary semaphores that include a priority inheritance mechanism. This means that if a high priority task blocks while attempting to obtain a mutex that is currently held by a lower priority task, the priority of the task holding the token is temporarily raised to that of the blocking task. This mechanism is designed to ensure the higher priority task is kept in the Blocked state for the shortest time possible, to minimize the priority inversion that has occurred.

Direct-to-task notifications

Task notifications allow tasks to interact with other tasks, and to synchronize with interrupt service routines (ISRs), without the need for a separate communication object like a semaphore. Each RTOS task has a 32-bit notification value that is used to store the content of the notification, if any. An RTOS task notification is an event sent directly to a task that can unblock the receiving task and optionally update the receiving task's notification value.

RTOS task notifications can be used as a faster and lightweight alternative to binary and counting semaphores and, in some cases, queues. Task notifications have both speed and RAM footprint advantages over other FreeRTOS features that can be used to perform equivalent functionality. However, task notifications can only be used when there is only one task that can be the recipient of the event.

Stream buffers

Stream buffers allow a stream of bytes to be passed from an interrupt service routine to a task, or from one task to another. A byte stream can be of arbitrary length and does not necessarily have a beginning or an end. Any number of bytes can be written at one time, and any number of bytes can be read at one time. You enable stream buffer functionality by including the `stream_buffer.c` source file in your project.

Stream buffers assume there is only one task or interrupt that writes to the buffer (the writer), and only one task or interrupt that reads from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupt service routines, but it is not safe to have multiple writers or readers.

The stream buffer implementation uses direct-to-task notifications. Therefore, calling a stream buffer API that places the calling task into the Blocked state can change the calling task's notification state and value.

Sending data

`xStreamBufferSend()` is used to send data to a stream buffer in a task. `xStreamBufferSendFromISR()` is used to send data to a stream buffer in an interrupt service routine (ISR).

`xStreamBufferSend()` allows a block time to be specified. If `xStreamBufferSend()` is called with a non-zero block time to write to a stream buffer and the buffer is full, the task is placed into the Blocked state until space becomes available or the block time expires.

`sbSEND_COMPLETED()` and `sbSEND_COMPLETED_FROM_ISR()` are macros that are called (internally, by the FreeRTOS API) when data is written to a stream buffer. It takes the handle of the stream buffer that was updated. Both of these macros check to see if there is a task blocked on the stream buffer waiting for data, and if so, removes the task from the Blocked state.

You can change this default behavior by providing your own implementation of `sbSEND_COMPLETED()` in `FreeRTOSConfig.h`. This is useful when a stream buffer is used to pass data between cores on a multicore processor. In that scenario, `sbSEND_COMPLETED()` can be implemented to generate an interrupt in the other CPU core, and the interrupt's service routine can then use the `xStreamBufferSendCompletedFromISR()` API to check, and if necessary unblock, a task that is waiting for the data.

Receiving data

`xStreamBufferReceive()` is used to read data from a stream buffer in a task. `xStreamBufferReceiveFromISR()` is used to read data from a stream buffer in an interrupt service routine (ISR).

`xStreamBufferReceive()` allows a block time to be specified. If `xStreamBufferReceive()` is called with a non-zero block time to read from a stream buffer and the buffer is empty, the task is placed into the Blocked state until either a specified amount of data becomes available in the stream buffer, or the block time expires.

The amount of data that must be in the stream buffer before a task is unblocked is called the stream buffer's trigger level. A task blocked with a trigger level of 10 is unblocked when at least 10 bytes are written to the buffer or the task's block time expires. If a reading task's block time expires before the trigger level is reached, the task receives any data written to the buffer. The trigger level of a task must be set to a value between 1 and the size of the stream buffer. The trigger level of a stream buffer is set when `xStreamBufferCreate()` is called. It can be changed by calling `xStreamBufferSetTriggerLevel()`.

`sbRECEIVE_COMPLETED()` and `sbRECEIVE_COMPLETED_FROM_ISR()` are macros that are called (internally, by the FreeRTOS API) when data is read from a stream buffer. The macros check to see if there is a task blocked on the stream buffer waiting for space to become available within the buffer, and if so, they remove the task from the Blocked state. You can change the default behavior of `sbRECEIVE_COMPLETED()` by providing an alternative implementation in `FreeRTOSConfig.h`.

Message buffers

Message buffers allow variable-length discrete messages to be passed from an interrupt service routine to a task, or from one task to another. For example, messages of length 10, 20, and 123 bytes can all be written to, and read from, the same message buffer. A 10-byte message can only be read as a 10-byte message, not as individual bytes. Message buffers are built on top of stream buffer implementation. you can enable message buffer functionality by including the `stream_buffer.c` source file in your project.

Message buffers assume there is only one task or interrupt that writes to the buffer (the writer), and only one task or interrupt that reads from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupt service routines, but it is not safe to have multiple writers or readers.

The message buffer implementation uses direct-to-task notifications. Therefore, calling a stream buffer API that places the calling task into the Blocked state can change the calling task's notification state and value.

To enable message buffers to handle variable-sized messages, the length of each message is written into the message buffer before the message itself. The length is stored in a variable of type `size_t`, which is typically 4 bytes on a 32-byte architecture. Therefore, writing a 10-byte message into a message buffer actually consumes 14 bytes of buffer space. Likewise, writing a 100-byte message into a message buffer actually uses 104 bytes of buffer space.

Sending data

`xMessageBufferSend()` is used to send data to a message buffer from a task. `xMessageBufferSendFromISR()` is used to send data to a message buffer from an interrupt service routine (ISR).

`xMessageBufferSend()` allows a block time to be specified. If `xMessageBufferSend()` is called with a non-zero block time to write to a message buffer and the buffer is full, the task is placed into the Blocked state until either space becomes available in the message buffer, or the block time expires.

`sbSEND_COMPLETED()` and `sbSEND_COMPLETED_FROM_ISR()` are macros that are called (internally, by the FreeRTOS API) when data is written to a stream buffer. It takes a single parameter, which is the handle of the stream buffer that was updated. Both of these macros check to see if there is a task blocked on the stream buffer waiting for data, and if so, they remove the task from the Blocked state.

You can change this default behavior by providing your own implementation of `sbSEND_COMPLETED()` in `FreeRTOSConfig.h`. This is useful when a stream buffer is used to pass data between cores on a multicore processor. In that scenario, `sbSEND_COMPLETED()` can be implemented to generate an interrupt in the other CPU core, and the interrupt's service routine can then use the `xStreamBufferSendCompletedFromISR()` API to check, and if necessary unblock, a task that was waiting for the data.

Receiving data

`xMessageBufferReceive()` is used to read data from a message buffer in a task. `xMessageBufferReceiveFromISR()` is used to read data from a message buffer in an interrupt service routine (ISR). `xMessageBufferReceive()` allows a block time to be specified. If

`xMessageBufferReceive()` is called with a non-zero block time to read from a message buffer and the buffer is empty, the task is placed into the Blocked state until either data becomes available, or the block time expires.

`sbRECEIVE_COMPLETED()` and `sbRECEIVE_COMPLETED_FROM_ISR()` are macros that are called (internally, by the FreeRTOS API) when data is read from a stream buffer. The macros check to see if there is a task blocked on the stream buffer waiting for space to become available within the buffer, and if so, they remove the task from the Blocked state. You can change the default behavior of `sbRECEIVE_COMPLETED()` by providing an alternative implementation in `FreeRTOSConfig.h`.

Symmetric multiprocessing (SMP) support

SMP support in the FreeRTOS Kernel enables one instance of the FreeRTOS kernel to schedule tasks across multiple identical processor cores. The core architectures must be identical and share the same memory.

The FreeRTOS API remains substantially the same between single-core and SMP versions, except for these additional APIs. Therefore, an application written for the FreeRTOS single-core version should compile with the SMP version with minimal to no effort. However, there might be some functional issues, because some assumptions that were true for single-core applications might no longer be true for multi-core applications.

One common assumption is that a lower priority task can't run while a higher priority task is running. While this was true on a single-core system, it's no longer true for multi-core systems because multiple tasks can run simultaneously. If the application relies on relative task priorities to provide mutual exclusion, it might observe unexpected results in a multi-core environment.

One other common assumption is that ISRs can't run simultaneously with each other or with other tasks. This is no longer true in a multi-core environment. The application writer needs to ensure proper mutual exclusion while accessing data shared between tasks and ISRs.

Software timers

A software timer allows a function to be executed at a set time in the future. The function executed by the timer is called the timer's *callback function*. The time between a timer being started and its callback function being executed is called the timer's *period*. The FreeRTOS kernel provides an efficient software timer implementation because:

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

Low power support

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

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

Configure the kernel (FreeRTOSConfig.h)

You can configure the FreeRTOS kernel for a specific board and application with the `FreeRTOSConfig.h` header file. Every application built on the kernel must have a `FreeRTOSConfig.h` header file in its preprocessor include path. `FreeRTOSConfig.h` is application-specific and should be placed under an application directory, and not in one of the FreeRTOS kernel source code directories.

The `FreeRTOSConfig.h` files for the FreeRTOS demo and test applications are located at *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/FreeRTOSConfig.h` and *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_tests/config_files/FreeRTOSConfig.h`.

For a list of the available configuration parameters to specify in `FreeRTOSConfig.h`, see FreeRTOS.org.

AWS IoT Device SDK for Embedded C

> **Note**
>
> This SDK is intended for use by experienced embedded-software developers.

The AWS IoT Device SDK for Embedded C (C-SDK) is a collection of C source files under the MIT open source license that can be used in embedded applications to securely connect IoT devices to AWS IoT Core. It includes an MQTT client, HTTP client, JSON Parser, and AWS IoT Device Shadow, AWS IoT Jobs, AWS IoT Fleet Provisioning, and AWS IoT Device Defender libraries. This SDK is distributed in source form and can be built into customer firmware along with application code, other libraries, and an operating system (OS) of your choice.

The AWS IoT Device SDK for Embedded C is generally targeted at resource constrained devices that require an optimized C language runtime. You can use the SDK on any operating system and host it on any processor type (for example, MCUs and MPUs). However, if your devices have sufficient memory and processing resources, we recommend that you use one of the higher order AWS IoT Device SDKs.

For more information, see the following:

- AWS IoT Device SDK for Embedded C
- AWS IoT Device SDK for Embedded C on GitHub
- AWS IoT Device SDK for Embedded C Readme
- AWS IoT Device SDK for Embedded C Samples

Understand the FreeRTOS Common IO APIs

Common IO APIs act as hardware abstraction layers (HAL) that provide a common interface between drivers and higher-level application code. FreeRTOS Common IO provides a set of standard APIs for accessing common serial devices on supported reference boards; implementations of these APIs are not included. These common APIs communicate and interact with these peripherals and enable your code to function across platforms. Without Common IO, writing code to work with low level devices is silicon-vendor specific.

> **Note**
>
> FreeRTOS does not require implementations of the Common IO APIs to function, but it will attempt to use the Common IO APIs as a way to interface with the specific peripherals on a microcontroller-based board instead of vendor-specific APIs.

In general, device drivers are independent of the underlying operating system and are specific to a given hardware configuration. The HAL abstracts away the details of how a specific driver works and provides a uniform API to control such devices. You can use the same APIs to access various device drivers across multiple microcontroller- (MCU-) based reference boards.

Libraries

Currently, FreeRTOS provides two Common IO libraries: Common IO - basic and Common IO - BLE.

Common IO - basic

Overview

Common IO - basic provides APIs that deal with basic I/O peripherals and functions that you may find on MCU-based boards. The Common IO - basic repository is available on GitHub.

Supported peripherals

- ADC
- GPIO
- I2C

- PWM
- SPI
- UART
- Watchdog
- Flash
- RTC
- EFUSE
- Resets
- I2S
- Performance counter
- Hardware platform information

Supported features

- Synchronous read/write

  The function doesn't return until the requested amount of data is transferred.

- Asynchronous read/write

  The function returns immediately and the data transfer happens asynchronously. When the action completes, a registered user callback is invoked.

Peripheral specific

- I2C

  Combine multiple operations into one transaction. Used to do write then read actions in one transaction.

- SPI

  Transfer data between primary and secondary, which means the write and read happen simultaneously.

API reference

For a full API reference, see the Common IO - basic API reference.

Common IO - BLE

Overview

Common IO - BLE provides abstraction from the manufacturer's Bluetooth Low Energy stack. It provides the following interfaces which can be used to control the device, and perform GAP and GATT operations. The Common IO - BLE repository is available on GitHub.

Bluetooth Device Manager:

This provides an interface to control the Bluetooth device, perform device discovery operations and other connectivity related tasks.

BLE Adapter Manager:

This provides an interface for the GAP API functions that are specific to BLE.

Bluetooth Classic Adapter Manager:

This provides an interface to control BT classic functionalities of a device.

GATT Server:

This provides an interface to use the Bluetooth GATT server feature.

GATT Client:

This provides an interface to use the Bluetooth GATT client feature.

A2DP Connection Interface:

This provides an interface for the A2DP Source profile for the local device.

API reference

For a full API reference, see the Common IO - BLE API reference.

Common IO for Amazon Common Software

The Common IO APIs are a part of the required implementations needed by Amazon Common Software for Devices, specifically to be implemented in a vendor device porting kit (DPK).

What is ACS?

Amazon Common Software (ACS) for Devices is software that makes it faster for you to integrate Amazon Device SDKs on your devices. ACS provides a unified API integration layer, pre-validated

and memory efficient components for common functions such as connectivity, a device porting kit (DPK), and multi-tier test suites.

Qualification Program

The Amazon Common Software for Devices qualification program verifies that a build of the ACS DPK (Device Porting Kit) which runs on a specific microcontroller-based development board is compatible with the program's published best practices and robust enough to pass ACS-mandated tests specified by the qualification program.

Vendors qualified under this program are listed on the ACS Chipset Vendors page.

For information about qualifying, contact ACS for Devices.

Get started with FreeRTOS

Topics

- Get started with Quick Connect
- Explore FreeRTOS libraries
- Build a secure and robust AWS IoT product
- Develop your AWS IoT application

Get started with Quick Connect

To quickly explore AWS IoT, start with AWS Quick Connect Demos. Quick Connect demos are simple to setup and connect a partner provided, FreeRTOS qualified board to AWS IoT.

Follow the AWS IoT Getting Started tutorial for a better understanding of AWS IoT and the AWS IoT console. You can modify the demo source code provided with the Quick Connect demos using the chosen board's build system and tools to connect to your AWS account. The data flow from the AWS IoT console on your account is visible now.

Explore FreeRTOS libraries

Once you have an understanding of how an IoT device and AWS IoT work together, you can start exploring FreeRTOS libraries, and the Long-Term Support (LTS) libraries.

Some commonly used libraries for FreeRTOS based AWS IoT devices are:

- FreeRTOS Kernel
- coreMQTT
- AWS IoT Over-the-Air (OTA)

Visit freertos.org for library-specific technical documentation and demos.

Build a secure and robust AWS IoT product

Refer to Featured FreeRTOS AWS IoT Integrations to learn about best practices in making IoT device software more secure and robust. These FreeRTOS IoT integrations are designed for

improved security using a combination of FreeRTOS software, and a partner-provided board with hardware security features. Use them in production as is, or use them as a model for your own designs.

Develop your AWS IoT application

Follow these steps to create an application project for your AWS IoT product:

1. Download the latest FreeRTOS or Long Term Support (LTS) version from freertos.org, or clone from the FreeRTOS-LTS GitHub repository. You can also integrate the required FreeRTOS libraries into your project from the MCU vendor's toolchain if available.
2. Follow the FreeRTOS Porting guide to create a project, set up the development environment, and integrate FreeRTOS libraries into your project. Use the FreeRTOS-Libraries-Integration-Tests GitHub repository to validate the porting.

AWS IoT Device Tester for FreeRTOS

The IDT for FreeRTOS is a tool to qualify data throughput rate with the FreeRTOS operating system. The device tester (IDT) first opens a USB or UART connection to a device. It then flashes an image of FreeRTOS configured to test the device functionality under various conditions. AWS IoT Device Tester suites are extensible and IDT is used for customer AWS IoT test orchestration.

IDT for FreeRTOS runs on a host computer (Windows, macOS, or Linux) that is connected to the device being tested. IDT configures and orchestrates test cases, and aggregates results. It also provides a command line interface to manage test execution.

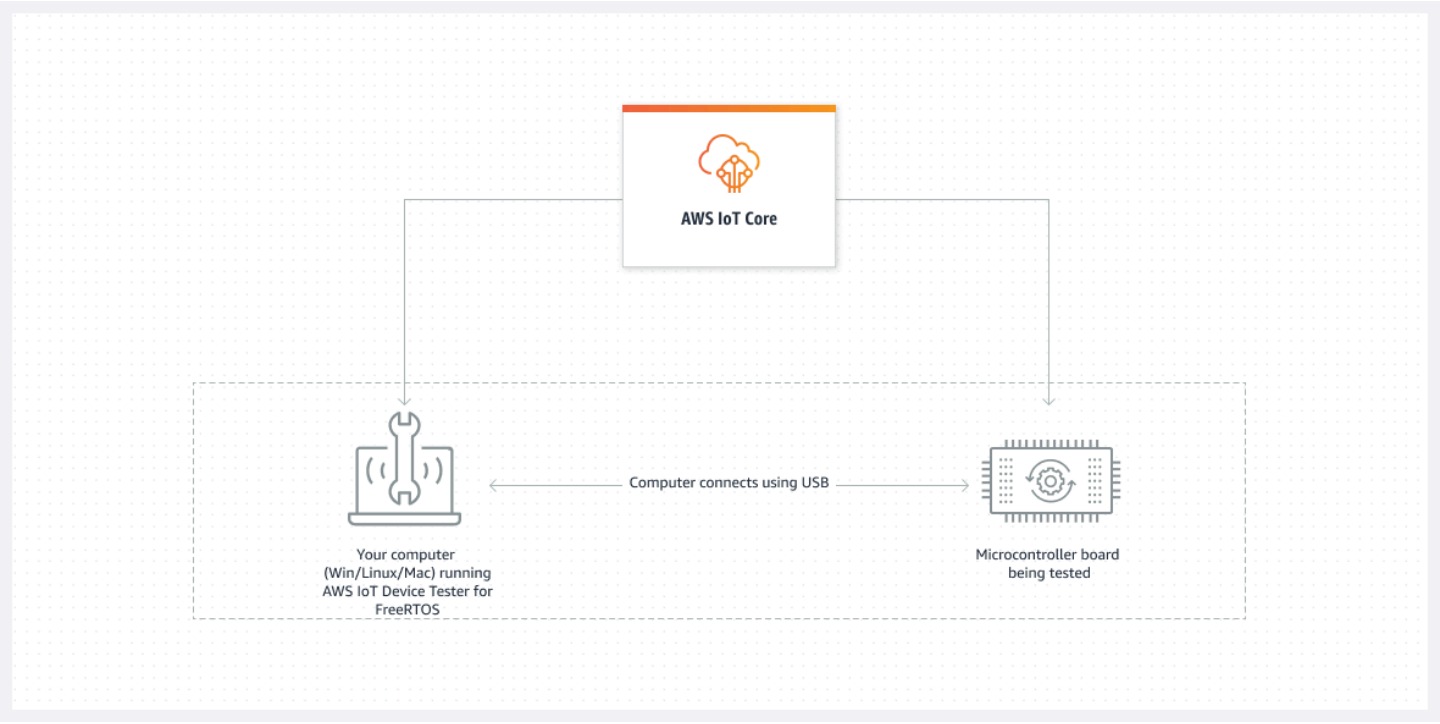
FreeRTOS qualification suite

IDT for FreeRTOS verifies the port of FreeRTOS on your micro-controller, and if it can communicate effectively with AWS IoT in a reliable and secure manner. Specifically, it verifies if the porting layer interfaces for FreeRTOS libraries are implemented correctly. It also performs end-to-end tests with AWS IoT Core. For example, it verifies if your board can send and receive MQTT messages and process them correctly.

FreeRTOS qualification (FRQ) 2.0 suite uses tests cases from FreeRTOS-Libraries-Integration-Tests and Device Advisor defined in the FreeRTOS Qualification Guide.

IDT for FreeRTOS generates test reports that you can submit to AWS Partner Network (APN) for inclusion of your FreeRTOS devices in the AWS Partner Device Catalog. For more information, see AWS Device Qualification Program.

The following diagram shows the test infrastructure setup for FreeRTOS qualification.



IDT for FreeRTOS organizes test resources into test suites and test groups:

- A test suite is the set of test groups used to verify that a device works with particular versions of FreeRTOS.
- A test group is the set of individual test cases related to a particular feature, such as BLE and MQTT messaging.

For more information, see Test suite versions

Understand custom test suites

IDT for FreeRTOS combines a standardized configuration setup and result format with a test suite environment. This environment lets you develop custom test suites for your devices and device software. You can add custom tests for your own internal validation, or provide them to your customers for device verification.

How you configure custom test suites determines the setting configurations that you must provide to your users to run your custom test suites. For more information, see Develop and run your own IDT test suites.

Supported versions of AWS IoT Device Tester

This topic lists supported versions of AWS IoT Device Tester for FreeRTOS. As a best practice, we recommend that you use the latest version of IDT for FreeRTOS that supports your target version of FreeRTOS. Each version of IDT for FreeRTOS has one or more corresponding versions of FreeRTOS that it supports. We recommend that you download a new version of IDT for FreeRTOS when a new version of FreeRTOS is released.

By downloading the software, you agree to the AWS IoT Device Tester License Agreement contained in the download archive.

Note

When you use AWS IoT Device Tester for FreeRTOS, we recommend that you update to the latest patch release of the most recent FreeRTOS-LTS version.

Important

As of October 2022, AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 doesn't generate signed qualification reports. You can't qualify new AWS IoT FreeRTOS devices to list in the AWS Partner Device Catalog through the AWS Device Qualification Program using IDT FRQ 1.0 versions. While you can't qualify FreeRTOS devices using IDT FRQ 1.0, you can continue to test your FreeRTOS devices with FRQ 1.0. We recommend that you use IDT FRQ 2.0 to qualify and list FreeRTOS devices in the AWS Partner Device Catalog.

Latest version of AWS IoT Device Tester for FreeRTOS

Use the following links to download the latest versions of IDT for FreeRTOS.

Latest version of AWS IoT Device Tester for FreeRTOS

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
IDT v4.9.0	FRQ_2.5.0	• 202112.00 • 202212.00 • 202212.01 • All patches of FreeRTOS 202210-LTS that use FreeRTOS LTS libraries.	• Linux • macOS • Windows	2023.04.04	• Supports testing against FreeRTOS **202112**, **202212**, **202212.01** and all patches of FreeRTOS **202210-LTS** that uses FreeRTOS libraries. See README.md for more information. You must include the patch version for FreeRTOS-LTS in your `manifest.yml` . • Improved run time of

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
					OTA E2E tests. • Limits number of devices listed in `device.json` to 1. • Minor bug fixes and improvements.

> **Note**
>
> We don't recommend that multiple users run IDT from a shared location, such as an NFS directory or a Windows network shared folder. This practice might result in crashes or data corruption. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Earlier IDT versions for FreeRTOS

The following earlier versions of IDT for FreeRTOS are also supported.

Earlier versions of AWS IoT Device Tester for FreeRTOS

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
IDT v4.8.1	FRQ_2.4.0	• 202112.00 • 202212.00 • 202212.01 • All patches of FreeRTOS 202210-LTS that use FreeRTOS LTS libraries.	• Linux • macOS • Windows	2023.01.23	• See README.MD for further information. You must include the patch version for FreeRTOS-LTS in your `manifest.yml` . • Minor bug fixes and improvements.
IDT v4.6.0	FRQ_2.3.0	• 202112.00 • 202212.00 • 202212.01 • 202210-LTS that use FreeRTOS LTS libraries.	• Linux • macOS • Windows	2022.11.16	• See README.MD for further information. You must include the patch version for FreeRTOS-LTS in your

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
					`manifest.yml` . • For more information about what's included in the FreeRTOS **202210-LTS** release, see the CHANGELOG.md file on GitHub. • Adds the ability to configure and run AWS IoT Device Tester for FreeRTOS through a web based user interface. See UI for IDT for FreeRTOS qualifica

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
					tion suite 2.0 (FRQ 2.0) to get started. • Adds an option to retain the modified copies of the source code created and used at runtime for post-test debugging. See Configure build, flash, and test settings for more information. • Adds IDT Client SDK support for Java. For more

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
					information about the IDT Client SDK, see Develop and run your own IDT test suites.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Download links	Release date	Release notes
IDT v4.5.11	FRQ_2.2.0	• 202112.00 • 202212.00 • 202212.01 • 202210-LTS that use FreeRTOS LTS libraries.	• Linux • macOS • Windows	2022.10.14	• See README.MD for further information. You must include the patch version for FreeRTOS-LTS in your `manifest.yml` . • For more information about what's included in the FreeRTOS **202210-LTS** release, see the CHANGELOG.md file on GitHub. • Minor bug fixes and improvements.

For more information, see Understand the support policy for AWS IoT Device Tester.

Unsupported IDT versions for FreeRTOS

This section lists unsupported versions of IDT for FreeRTOS. Unsupported versions do not receive bug fixes or updates. For more information, see Understand the support policy for AWS IoT Device Tester.

The following versions of IDT-FreeRTOS are no longer supported.

Unsupported versions of AWS IoT Device Tester for FreeRTOS

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.5.10	FRQ_2.1.4	• 202112.00 • 202012-LTS that use FreeRTOS LTS libraries.	2022.09.02	• For more information about what's included in the FreeRTOS **202012-LTS** release, see the CHANGELOG.md file on GitHub. • Resolved an issue affecting the `OTA End to End` test group. • Removed `FullTransportInterfacePlainText` from running in

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				qualification runs. Plain text can still be ran as a development test group by using the `-\-group-id` flag. • Improved the logging and the readability of console and file output. • Minor bug fixes and improveme nts.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.5.9	FRQ_2.1.3	• 202112.00 • 202012.04-LTS that use FreeRTOS LTS libraries.	2022.08.17	• For more information about what's included in the FreeRTOS 202012.04-LTS release, see the CHANGELOG.md file on GitHub. • Resolved an issue affecting the `FreeRTOSIntegrity` test group. • Updated `FullCloudIoT` test group by removing “MQTT Connect Exponential Backoff Retries” test case. • Minor bug fixes and

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				improvements.
IDT v4.5.6	FRQ_2.1.2	• 202112.00 • 202012.04-LTS that use FreeRTOS LTS libraries.	2022.06.29	• For more information about what's included in the FreeRTOS 202012.04-LTS release, see the CHANGELOG.md file on GitHub. • Adds new test group `FullCloudIoT` which tests the board against AWS IoT Core Device Advisor. • Resolved an issue affecting the OTA E2E test cases. • Minor bug fixes and improvements.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.5.5	FRQ_2.1.1	• 202112.00 • 202012.04-LTS that use FreeRTOS LTS libraries.	2022.06.06	• For more information about what's included in the FreeRTOS 202012.04-LTS release, see the CHANGELOG.md file on GitHub. • Adds new test group `FullCloudIoT` which tests the board against AWS IoT Core Device Advisor. • Resolved an issue affecting the FreeRTOSVersion and FreeRTOSIntegrity test cases. • Minor bug fixes and

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				improvements.
IDT v4.5.5	FRQ_2.1.0	• 202107.00 • 202112.00 • 202012.04-LTS that use FreeRTOS LTS libraries.	2022.05.31	• For more information about what's included in the FreeRTOS 202012.04-LTS release, see the CHANGELOG.md file on GitHub. • Adds new test group `FullCloudIoT` which tests the board against AWS IoT Core Device Advisor. • Minor bug fixes and improvements.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.5.4	FRQ_2.0.0	• 202112.00 • 202012.04-LTS that use FreeRTOS LTS libraries.	2022.05.09	• For more information about what's included in the FreeRTOS 202012.04-LTS release, see the CHANGELOG.md file on GitHub. • Removes the requirement to qualify boards using only versions of Amazon FreeRTOS from the `aws/amazon-freertos` GitHub repository. • Minor bug fixes and improvements.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.5.2	FRQ_1.6.2	202107.00	2022.01.25	• For more information about what's included in the FreeRTOS 202107.00 release, see the CHANGELOG.md file on GitHub. • Implements the new IDT test orchestrator for configuring custom test suites. For more information, see Configure the IDT test orchestrator. • Minor bug fixes and improvements.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.0.3	FRQ_1.5.1	202012.00	2021.07.30	• Support for qualification of devices with locked-down credentials on a Hardware Security Module. • Minor bug fixes and improvements.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.3.0	FRQ_1.6.1	202107.00	2021.07.26	• For more information about what's included in the FreeRTOS 202107.00 release, see the CHANGELOG.md file on GitHub. • Adds the ability to configure and run AWS IoT Device Tester for FreeRTOS through a web based user interface. See Use the IDT user interface to run the FreeRTOS qualification suite to get started.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes

IDT v4.1.0	FRQ_1.6.0	202107.00	2021.07.21	• For more information about what's included in the FreeRTOS 202107.00 release, see the CHANGELOG.md file on GitHub.
				• Removes the following test cases from OTA qualification:
				  • OTA Agent
				  • OTA Missing Filename
				  • OTA Max Configured Number of Blocks
				• Removes the OTA Dataplane Both test group from OTA Qualification. In the device.js

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				on file, the `OTADataPlaneProtocol` configuration now accepts only `HTTP` or `MQTT` as supported values. • Implements the following changes to the `freertosFileConfiguration` configuration in the `userdata.json` file for changes to the FreeRTOS source code: • Changes the file name that is specified for `otaAgentTestsConfig` and `otaAgentDemosConfig` from

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				`aws_ota_agent_config.h` to `ota_config.h`. • Adds a new `otaDemosConfig` optional configuration to specify the file path to the new `ota_demo_config.h` file. • Adds a new field `testStartDelayms` to `userdata.json` to specify a delay between the time a device is flashed to run a FreeRTOS test group and when it starts running

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				tests. The value should be given in milliseconds. This delay can be used to give IDT a chance to connect so that no test output is missed.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v4.0.1	FRQ_1.4.1	202012.00	2021.01.19	• For more information about what's included in the FreeRTOS 202012.00 release, see the CHANGELOG.md file in GitHub. • Introduces additional OTA (Over-the-air) E2E (end-to-end) test cases. • Supports qualification of development boards running FreeRTOS 202012.00 that use FreeRTOS LTS libraries. • Adds support for qualification of FreeRTOS developme

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				nt boards using cellular connectivity. • Fixes a bug in the echo server configuration. • Enables you to develop and run your own custom test suites using AWS IoT Device Tester for FreeRTOS. For more information, see Develop and run your own IDT test suites. • Provides code signed IDT applications, so you don't need to grant permissions when you run it under Windows or macOS.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				• Refined the BLE test result parsing logic.
IDT v3.4.0	FRQ_1.3.0	202011.01	2020.11.05	• For more details, see the CHANGELOG.md file in GitHub. • Fixed bug where 'RSA' was not a valid PKCS11 configuration option. • Fixed bug where Amazon S3 buckets aren't cleaned up correctly after OTA tests. • Updates to support the new test cases inside of the FullMQTT test group.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v3.3.0	FRQ_1.2.0	202007.00	2020.09.17	• For more details, see the CHANGELOG.md file in GitHub. • New end-to-end tests to validate Over The Air (OTA) update suspend and resume feature. • Fixed bug causing users in eu-central-1 Region to be unable to pass config validation for OTA tests. • Added `--update-idt` parameter to the `run-suite` command. You can use this option to set the response

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				for the IDT update prompt. • Added `--update-managed-policy` parameter to the `run-suite` command. You can use this option to set the response for the managed policy update prompt. • Internal improvements and bug fixes, including: • For automatic test suite updates, improvements to config file upgrade.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes

IDT v3.0.2	FRQ_1.0.1	202002.00		• For more information, see the CHANGELOG.md file in GitHub.
				• Adds automatic update of test suites within IDT. IDT can now download the latest test suites that are available for your FreeRTOS version. With this feature, you can:
				  • Download the latest test suites using the `upgrade-test-suite` command.
				  • Download the latest test suites by setting

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				a flag when you start IDT. Use the `-u` *`flag`* option where *`flag`* can be 'y' to always download or 'n' to use the existing version. When there are multiple test suite versions available, the latest version is used unless you specify a test suite ID when starting IDT. • Use the new `list-supported-versions` option to list the FreeRTOS

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				and test suite versions that are supported by the installed version of IDT. • List test cases in a group and run individual tests. Test suites are versioned using a `major.minor.patch` format starting from 1.0.0. • Adds the `list-supported-products` command – Lists the FreeRTOS and test suite versions that are

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				supported by the installed version of IDT. • Adds `list-test-cases` command – Lists the test cases that are available in a test group. • Adds the `test-id` option for the `run-suite` command – Use this option to run individual test cases in a test group.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v1.7.1	FRQ_1.0.0	202002.00		• For more details, see the CHANGELOG.md file in GitHub. • Supports the custom code signing method for over-the-air (OTA) end-to-end test cases so that you can use your own code signing commands and scripts to sign OTA payloads. • Adds a precheck for serial ports before the start of tests. Tests will fail quickly with improved error messaging if the serial port

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				is misconfig ured in the `device.js on` file. • Added an AWS Managed Policy `AWSIoTDev iceTester ForFreeRT OSFullAcc ess` with permissions required to run AWS IoT Device Tester. If new releases require additiona l permissio ns, we add them to this managed policy so that you don't have to manually update your IAM permissio ns. • The file named `AFQ_Repor`

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
				`t.xml` in the results directory is now `FRQ_Repor t.xml` .
IDT v1.6.2	FRQ_1.0.0	202002.00		• Supports optional tests for OTA over HTTPS to qualify your FreeRTOS development boards. • Supports AWS IoT ATS endpoint in testing. • Supports capability to inform users on latest IDT version before start of test suite.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v1.5.2	FRQ_1.0.0	201910.00		• Supports qualification of FreeRTOS devices with secure element (onboard key). • Supports configura ble echo server ports for Secure Sockets and Wi-Fi test groups. • Supports timeout multiplier flag to increase timeouts, which is useful when you troubleshoot for timeout-r elated errors. • Added bug fix for log parsing. • Supports iot ats endpoint in testing.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT v1.4.1	FRQ_1.0.0	201908.00		• Added support for new PKCS11 library and test case updates. • Introduced actionable error codes. For more information, see IDT error codes • Updated IAM policy used to run IDT.
IDT v1.3.2	FRQ_1.0.0	201906.00		• Added support for testing Bluetooth Low Energy (BLE). • Improved user experienc e for IDT command line interface (CLI) commands. • Updated IAM policy used to run IDT.

AWS IoT Device Tester version	Test suite versions	Supported FreeRTOS versions	Release date	Release notes
IDT-FreeRTOS v1.2	FRQ_1.0.0	• FreeRTOS v1.4.8 • FreeRTOS v1.4.9		Added support for testing FreeRTOS devices with the CMAKE build system.
IDT-FreeRTOS v1.1	FRQ_1.0.0			
IDT-FreeRTOS v1.0	FRQ_1.0.0			

Download IDT for FreeRTOS

This topic describes the options to download IDT for FreeRTOS. You can either use one of the following software download links or you can follow instructions to programmatically download IDT.

Important

As of October 2022, AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 does not generate signed qualification reports. You cannot qualify new AWS IoT FreeRTOS devices to list in the AWS Partner Device Catalog through the AWS Device Qualification Program using IDT FRQ 1.0 versions. While you can't qualify FreeRTOS devices using IDT FRQ 1.0, you can continue to test your FreeRTOS devices with FRQ 1.0. We recommend that you use IDT FRQ 2.0 to qualify and list FreeRTOS devices in the AWS Partner Device Catalog.

Topics

- Download IDT manually
- Download IDT programmatically

By downloading the software, you agree to the AWS IoT Device Tester License Agreement contained in the download archive.

> **Note**
>
> IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Download IDT manually

This topic lists supported versions of IDT for FreeRTOS. As a best practice, we recommend that you use the latest version of AWS IoT Device Tester that supports your target version of FreeRTOS. New releases of FreeRTOS might require you to download a new version of AWS IoT Device Tester. You receive a notification when you start a test run if AWS IoT Device Tester is not compatible with the version of FreeRTOS you are using.

See Supported versions of AWS IoT Device Tester

Download IDT programmatically

IDT provides an API operation that you can use to retrieve a URL where you can download IDT programmatically. You can also use this API operation to check if you have the latest version of IDT. This API operation has the following endpoint.

```
https://download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt
```

To call this API operation, you must have permission to perform the **`iot-device-tester:LatestIdt`** action. Include your AWS signature, with `iot-device-tester` as the Service Name

API request

HostOs – The operating system of the host machine. Choose from the following options:

- `mac`
- `linux`
- `windows`

TestSuiteType – The type of the test suite. Choose the following option:

FR – IDT for FreeRTOS

ProductVersion

(Optional) The version of FreeRTOS. The service returns the latest compatible version of IDT for that version of FreeRTOS. If you don't specify this option, the service returns the latest version of IDT.

API response

The API response has the following format. The `DownloadURL` includes a zip file.

```
{
    "Success": True or False,
    "Message": Message,
    "LatestBk": {
        "Version": The version of the IDT binary,
        "TestSuiteVersion": The version of the test suite,
        "DownloadURL": The URL to download the IDT Bundle, valid for one hour
    }
 }
```

Examples

You can reference the following examples to programmatically download IDT. These examples use credentials that you store in the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables. To follow best security practices, don't store your credentials in your code.

Example

Example: Download using cURL version 7.75.0 or later (Mac and Linux)

If you have cURL version 7.75.0 or later, you can use the `aws-sigv4` flag to sign the API request. This example uses jq to parse the download URL from the response.

> **Warning**
>
> The `aws-sigv4` flag requires the query parameters of the curl GET request be in the order of **HostOs/ProductVersion/TestSuiteType** or **HostOs/TestSuiteType**. Orders that do not

conform, will result in an error of getting mismatched signatures for the Canonical String from the API Gateway.
If the optional parameter **ProductVersion** is included, you must use a supported product version as documented in Supported versions of AWS IoT Device Tester for FreeRTOS.

- Replace *us-west-2* with your AWS Region. For the list of Region codes, see Regional endpoints.
- Replace *linux* with your host machine's operating system.
- Replace *202107.00* with your version of FreeRTOS.

```
url=$(curl --request GET "https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&ProductVersion=202107.00&TestSuiteType=FR" \
--user $AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY \
--aws-sigv4 "aws:amz:us-west-2:iot-device-tester" \
| jq -r '.LatestBk["DownloadURL"]')

curl $url --output devicetester.zip
```

Example

Example: Download using an earlier version of cURL (Mac and Linux)

You can use the following cURL command with an AWS signature that you sign and calculate. For more information about how to sign and calculate an AWS signature, see Signing AWS API requests.

- Replace *linux* with your host machine's operating system.
- Replace *Timestamp* with the date and time, such as **20220210T004606Z**.
- Replace *Date* with the date, such as **20220210**.
- Replace *AWSRegion* with your AWS Region. For the list of Region codes, see Regional endpoints.
- Replace *AWSSignature* with the AWS signature that you generate.

```
curl --location --request GET 'https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&TestSuiteType=FR' \
```

```
--header 'X-Amz-Date: Timestamp \
--header 'Authorization: AWS4-HMAC-SHA256 Credential=$AWS_ACCESS_KEY_ID/Date/AWSRegion/
iot-device-tester/aws4_request, SignedHeaders=host;x-amz-date, Signature=AWSSignature'
```

Example

Example: Download using a Python script

This example uses the Python requests library. This example is adapted from the Python example to Sign an AWS API request in the *AWS General Reference*.

- Replace *us-west-2* with your region. For the list of Region codes, see Regional endpoints.
- Replace *linux* with your host machine's operating system.

```
# Copyright 2010-2022 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of the
#License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
# OF ANY KIND, either express or implied. See the License for the specific
# language governing permissions and limitations under the License.

# See: http://docs.aws.amazon.com/general/latest/gr/sigv4_signing.html
# This version makes a GET request and passes the signature
# in the Authorization header.
import sys, os, base64, datetime, hashlib, hmac
import requests # pip install requests
# ************* REQUEST VALUES *************
method = 'GET'
service = 'iot-device-tester'
host = 'download.devicetester.iotdevicesecosystem.amazonaws.com'
region = 'us-west-2'
endpoint = 'https://download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt'
request_parameters = 'HostOs=linux&TestSuiteType=FR'

# Key derivation functions. See:
# http://docs.aws.amazon.com/general/latest/gr/signature-v4-examples.html#signature-v4-
examples-python
```

```
def sign(key, msg):
    return hmac.new(key, msg.encode('utf-8'), hashlib.sha256).digest()

def getSignatureKey(key, dateStamp, regionName, serviceName):
    kDate = sign(('AWS4' + key).encode('utf-8'), dateStamp)
    kRegion = sign(kDate, regionName)
    kService = sign(kRegion, serviceName)
    kSigning = sign(kService, 'aws4_request')
    return kSigning

# Read AWS access key from env. variables or configuration file. Best practice is NOT
# to embed credentials in code.
access_key = os.environ.get('AWS_ACCESS_KEY_ID')
secret_key = os.environ.get('AWS_SECRET_ACCESS_KEY')
if access_key is None or secret_key is None:
    print('No access key is available.')
    sys.exit()

# Create a date for headers and the credential string
t = datetime.datetime.utcnow()
amzdate = t.strftime('%Y%m%dT%H%M%SZ')
datestamp = t.strftime('%Y%m%d') # Date w/o time, used in credential scope

# ************* TASK 1: CREATE A CANONICAL REQUEST *************
# http://docs.aws.amazon.com/general/latest/gr/sigv4-create-canonical-request.html
# Step 1 is to define the verb (GET, POST, etc.)--already done.
# Step 2: Create canonical URI--the part of the URI from domain to query
# string (use '/' if no path)
canonical_uri = '/latestidt'
# Step 3: Create the canonical query string. In this example (a GET request),
# request parameters are in the query string. Query string values must
# be URL-encoded (space=%20). The parameters must be sorted by name.
# For this example, the query string is pre-formatted in the request_parameters
 variable.
canonical_querystring = request_parameters
# Step 4: Create the canonical headers and signed headers. Header names
# must be trimmed and lowercase, and sorted in code point order from
# low to high. Note that there is a trailing \n.
canonical_headers = 'host:' + host + '\n' + 'x-amz-date:' + amzdate + '\n'
# Step 5: Create the list of signed headers. This lists the headers
# in the canonical_headers list, delimited with ";" and in alpha order.
# Note: The request can include any headers; canonical_headers and
# signed_headers lists those that you want to be included in the
# hash of the request. "Host" and "x-amz-date" are always required.
```

```
signed_headers = 'host;x-amz-date'
# Step 6: Create payload hash (hash of the request body content). For GET
# requests, the payload is an empty string ("").
payload_hash = hashlib.sha256(('').encode('utf-8')).hexdigest()
# Step 7: Combine elements to create canonical request
canonical_request = method + '\n' + canonical_uri + '\n' + canonical_querystring + '\n'
 + canonical_headers + '\n' + signed_headers + '\n' + payload_hash

# ************* TASK 2: CREATE THE STRING TO SIGN*************
# Match the algorithm to the hashing algorithm you use, either SHA-1 or
# SHA-256 (recommended)
algorithm = 'AWS4-HMAC-SHA256'
credential_scope = datestamp + '/' + region + '/' + service + '/' + 'aws4_request'
string_to_sign = algorithm + '\n' +  amzdate + '\n' +  credential_scope + '\n' +
 hashlib.sha256(canonical_request.encode('utf-8')).hexdigest()
# ************* TASK 3: CALCULATE THE SIGNATURE *************
# Create the signing key using the function defined above.
signing_key = getSignatureKey(secret_key, datestamp, region, service)
# Sign the string_to_sign using the signing_key
signature = hmac.new(signing_key, (string_to_sign).encode('utf-8'),
 hashlib.sha256).hexdigest()

# ************* TASK 4: ADD SIGNING INFORMATION TO THE REQUEST *************
# The signing information can be either in a query string value or in
# a header named Authorization. This code shows how to use a header.
# Create authorization header and add to request headers
authorization_header = algorithm + ' ' + 'Credential=' + access_key + '/' +
 credential_scope + ', ' +  'SignedHeaders=' + signed_headers + ', ' + 'Signature=' +
 signature
# The request can include any headers, but MUST include "host", "x-amz-date",
# and (for this scenario) "Authorization". "host" and "x-amz-date" must
# be included in the canonical_headers and signed_headers, as noted
# earlier. Order here is not significant.
# Python note: The 'host' header is added automatically by the Python 'requests'
 library.
headers = {'x-amz-date':amzdate, 'Authorization':authorization_header}

# ************* SEND THE REQUEST *************
request_url = endpoint + '?' + canonical_querystring
print('\nBEGIN REQUEST++++++++++++++++++++++++++++++++++++')
print('Request URL = ' + request_url)
response = requests.get(request_url, headers=headers)
print('\nRESPONSE++++++++++++++++++++++++++++++++++++')
print('Response code: %d\n' % response.status_code)
```

```
print(response.text)

download_url = response.json()["LatestBk"]["DownloadURL"]
r = requests.get(download_url)
open('devicetester.zip', 'wb').write(r.content)
```

IDT with FreeRTOS qualification suite 2.0 (FRQ 2.0)

The FreeRTOS qualification suite 2.0 is an updated version of FreeRTOS qualification suite. We recommend developers to use FRQ 2.0 because it consists of relevant test cases to qualify devices that run FreeRTOS Long Term Support (LTS) libraries.

IDT for FreeRTOS verifies the port of FreeRTOS on your micro-controller, and if it communicates effectively with AWS IoT. Specifically, it verifies the porting layer interfaces with the FreeRTOS libraries, and if FreeRTOS test repositories are implemented correctly. It also performs end-to-end tests with AWS IoT Core. The tests run by IDT for FreeRTOS are defined in the FreeRTOS GitHub repository.

IDT for FreeRTOS run tests as embedded applications that it flashes on the microcontroller device under test. The application binary images include FreeRTOS, the ported FreeRTOS interfaces, and board device drivers. The purpose of the tests is to verify that the ported FreeRTOS interfaces function correctly on top of your device drivers.

IDT for FreeRTOS generates test reports that you can submit to AWS IoT to get your hardware listed on the AWS Partner Device Catalog. For more information, see AWS Device Qualification Program.

IDT for FreeRTOS runs on a host computer (Windows, macOS, or Linux) that is connected to the device under testing. IDT configures and orchestrates test cases and aggregates results. It also provides a command line interface to manage running the tests.

In order to test your device, IDT for FreeRTOS creates resources such as AWS IoT things, FreeRTOS groups, Lambda functions. To create these resources, IDT for FreeRTOS uses the AWS credentials configured in the `config.json` to make API calls on your behalf. These resources are provisioned at various times during a test.

When you run IDT for FreeRTOS on your host computer, it performs the following steps:

1. Loads and validates your device and credentials configuration.

2. Performs selected tests with the required local and cloud resources.
3. Cleans up local and cloud resources.
4. Generates tests reports that indicate if your board passed the tests required for qualification.

Set up the LTS qualification prerequisites

This section describes the prerequisites for testing microcontrollers with AWS IoT Device Tester.

Prepare for FreeRTOS qualification

> **Note**
>
> AWS IoT Device Tester for FreeRTOS strongly recommends using the latest patch release of the most recently FreeRTOS-LTS version.

IDT for FRQ 2.0 is a qualification for FreeRTOS. Before running IDT FRQ 2.0 for qualification, you must complete Qualifying your board in the *FreeRTOS Qualification Guide*. To port libraries, tests, and setup the `manifest.yml`, see Porting the FreeRTOS libraries in the *FreeRTOS Porting Guide*. FRQ 2.0 contains a different process for qualification. See Latest changes in qualification in the *FreeRTOS qualification guide* for details.

The FreeRTOS-Libraries-Integration-Tests repository must be present for IDT to run. See the README.md on how to clone and port this repository to your source project. FreeRTOS-Libraries-Integration-Tests must include the `manifest.yml` located in the root of your project, for IDT to run.

> **Note**
>
> IDT is dependent on the tests repository's implementation of `UNITY_OUTPUT_CHAR`. The test output logs and the device logs must not interleave with each other. See Implementing the library logging macros section in the *FreeRTOS Porting Guide* for further details.

Download IDT for FreeRTOS

Every version of FreeRTOS has a corresponding version of IDT for FreeRTOS to perform qualification tests. Download the appropriate version of IDT for FreeRTOS from Supported versions of AWS IoT Device Tester for FreeRTOS.

Extract IDT for FreeRTOS to a location on the file system where you have read and write permissions. Since Microsoft Windows has a character limit for the path length, extract IDT for FreeRTOS into a root directory such as `C:\` or `D:\`.

> **Note**
>
> Multiple users must not run IDT from a shared location, such as an NFS directory or a Windows network shared folder. This will result in crashes or data corruption. We recommend that you extract the IDT package to a local drive.

Download Git

IDT must have Git installed as a prerequisite to ensure source code integrity.

Follow the instructions in the GitHub guide to install Git. To verify the current installed version of Git, enter the command `git --version` at the terminal.

> **Warning**
>
> IDT uses Git to align with a directory's status of clean or dirty. If Git is not installed, the `FreeRTOSIntegrity` test groups will either fail, or won't run as expected. If IDT returns an error such as `git executable not found` or `git command not found`, install or re-install Git and try again.

Topics

- Create an AWS account
- AWS IoT Device Tester managed policy
- (Optional) Install the AWS Command Line Interface

Create an AWS account

> **Note**
>
> The full IDT qualification suite is supported only in the following AWS Regions
>
> - US East (N. Virginia)
> - US West (Oregon)
> - Asia Pacific (Tokyo)
> - Europe (Ireland)

In order to test your device, IDT for FreeRTOS creates resources like AWS IoT things, FreeRTOS groups and Lambda functions. To create those resources, IDT for FreeRTOS requires you to create and configure an AWS account, and an IAM policy that grants IDT for FreeRTOS permission to access resources on your behalf while running tests.

The following steps are to create and configure your AWS account.

1. If you already have an AWS account, skip to the next step. Else create an AWS account.
2. Follow the steps in Creating IAM roles. Do not add permissions or policies at this time.
3. To run OTA qualification tests, go to Step 4. Else go to Step 5.
4. Attach the OTA IAM permissions inline policy to your IAM role.

   a.

   > **Important**
   >
   > The following policy template grants IDT permission to create roles, create policies, and attach policies to roles. IDT for FreeRTOS uses these permissions for tests that create roles. Although the policy template doesn't provide administrator privileges to the user, the permissions can be used to gain administrator access to your AWS account.

   b. Follow the steps below to attach the necessary permissions to your IAM role:

      i. On the **Permissions** page, choose **Add permissions**.

      ii. Choose **Create inline policy**.

iii. Choose the **JSON** tab and copy the following permissions in to the **JSON** text box. Use the template under **Most Regions** if you are not in the China region. If you are in the China region, use the template under **Beijing and Ningxia Regions**.

Most Regions

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iotdeviceadvisor:*",
            "Resource": [
                "arn:aws:iotdeviceadvisor:*:*:suiterun/*/*",
                "arn:aws:iotdeviceadvisor:*:*:suitedefinition/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::*:role/idt*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService":
 "iotdeviceadvisor.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "execute-api:Invoke*",
                "iam:ListRoles",
                "iot:Connect",
                "iot:CreateJob",
                "iot:DeleteJob",
                "iot:DescribeCertificate",
                "iot:DescribeEndpoint",
                "iot:DescribeJobExecution",
```

```
                "iot:DescribeJob",

                "iot:DescribeThing",
                "iot:GetPolicy",
                "iot:ListAttachedPolicies",
                "iot:ListCertificates",
                "iot:ListPrincipalPolicies",
                "iot:ListThingPrincipals",
                "iot:ListThings",
                "iot:Publish",
                "iot:UpdateThingShadow",
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:PutRetentionPolicy"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "iotdeviceadvisor:*",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "logs:DeleteLogGroup",
            "Resource": "arn:aws:logs:*:*:log-group:/aws/iot/
deviceadvisor/*"
        },
        {
            "Effect": "Allow",
            "Action": "logs:GetLogEvents",
            "Resource": "arn:aws:logs:*:*:log-group:/aws/iot/
deviceadvisor/*:log-stream:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreatePolicy",
                "iam:DetachRolePolicy",
                "iam:DeleteRolePolicy",
                "iam:DeletePolicy",
```

```
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:AttachRolePolicy"
            ],
            "Resource": [
                "arn:aws:iam::*:policy/idt*",
                "arn:aws:iam::*:role/idt*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ssm:GetParameters"
            ],
            "Resource": [
                "arn:aws:ssm:*::parameter/aws/service/ami-amazon-
linux-latest/amzn2-ami-hvm-x86_64-gp2"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:RunInstances",
                "ec2:CreateSecurityGroup",
                "ec2:CreateTags",
                "ec2:DeleteTags"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateKeyPair",
                "ec2:DeleteKeyPair"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:key-pair/idt-ec2-ssh-key-*"
            ]
        },
        {
            "Effect": "Allow",
```

```
            "Condition": {
                "StringEqualsIgnoreCase": {
                    "aws:ResourceTag/Owner": "IoTDeviceTester"
                }
            },
            "Action": [
                "ec2:TerminateInstances",
                "ec2:DeleteSecurityGroup",
                "ec2:AuthorizeSecurityGroupIngress",
                "ec2:RevokeSecurityGroupIngress"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Beijing and Ningxia Regions

The following policy template can be used in the Beijing and Ningxia Regions.

iv. When you're finished, choose **Review policy**.

v. Enter **IDTFreeRTOSIAMPermissions** as the policy name.

vi. Choose **Create policy**.

5. Attach **AWSIoTDeviceTesterForFreeRTOSFullAccess** to your IAM role.

   a. To attach the necessary permissions to your IAM role:

      i. On the **Permissions** page, choose **Add permissions**.

      ii. Choose **Attach policies**.

      iii. Search for the **AWSIoTDeviceTesterForFreeRTOSFullAccess** policy. Check the box.

   b. Choose **Add permissions**.

6. Export credentials for IDT. See Getting IAM role credentials for CLI access for details.

AWS IoT Device Tester managed policy

The `AWSIoTDeviceTesterForFreeRTOSFullAccess` managed policy contains the following AWS IoT Device Tester permissions for version checking, auto update features, and collection of metrics.

- `iot-device-tester:SupportedVersion`

  Grants AWS IoT Device Tester permission to fetch the list of supported products, test suites and IDT versions.
- `iot-device-tester:LatestIdt`

  Grants AWS IoT Device Tester permission to fetch the latest IDT version available for download.
- `iot-device-tester:CheckVersion`

  Grants AWS IoT Device Tester permission to check version compatibility for IDT, test suites and products.
- `iot-device-tester:DownloadTestSuite`

  Grants AWS IoT Device Tester permission to download test suite updates.
- `iot-device-tester:SendMetrics`

  Grants AWS permission to collect metrics about AWS IoT Device Tester internal usage.

(Optional) Install the AWS Command Line Interface

You might prefer to use the AWS CLI to perform some operations. If you don't have the AWS CLI installed, follow the instructions at Install the AWS CLI.

Configure the AWS CLI for the AWS Region you want to use by running **aws configure** from a command line. For information about the AWS Regions that support IDT for FreeRTOS, see AWS Regions and Endpoints. For more information about **aws configure** see Quick configuration with **aws configure**.

First test of your microcontroller board

You can use IDT for FreeRTOS to test your implementation of the FreeRTOS libraries. After you have ported the FreeRTOS libraries for your board's device drivers, use AWS IoT Device Tester to run the qualification tests on your microcontroller board.

Add library porting layers

To port FreeRTOS for your device, see the FreeRTOS Porting Guide. When implementing the FreeRTOS tests repository and porting the FreeRTOS layers, you must provide a `manifest.yml` with paths to each library, including the tests repository. This file will be in the root directory of your source code. See manifest file instructions for details.

Configure your AWS credentials for AWS IoT Device Tester to communicate with the AWS Cloud

You need to configure your AWS credentials for AWS IoT Device Tester to communicate with the AWS Cloud. For more information, see Set up AWS Credentials and Region for Development. Valid AWS credentials are specified in the *devicetester_extract_location*/devicetester_freertos_*[win|mac|linux]*/configs/config.json configuration file.

```
"auth": {
   "method": "environment"
}

"auth": {
    "method": "file",
    "credentials": {
        "profile": "<your-aws-profile>"
    }
}
```

The `auth` attribute of the `config.json` file has a method field that controls AWS authentication, and can be declared as either file or environment. Setting the field to environment pulls your AWS credentials from your host machine's environment variables. Setting the field to file imports a specified profile from the `.aws/credentials` config file.

Topics

- Create a device pool in IDT for FreeRTOS
- Configure build, flash, and test settings

Create a device pool in IDT for FreeRTOS

Devices to be tested are organized in device pools. Each device pool consists of one or more identical devices. You can configure IDT for FreeRTOS to test a single device, or multiple devices in a pool. To accelerate the qualification process, IDT for FreeRTOS can test devices with the same specifications in parallel. It uses a round-robin method to execute a different test group on each device in a device pool.

The `device.json` file has an array in its top level. Each array attribute is a new device pool. Each device pool has a devices array attribute, which has multiple devices declared. In the template, there is a device pool and only one device in that device pool. You can add one or more devices to a device pool by editing the `devices` section of the `device.json` template in the `configs` folder.

> **Note**
>
> All devices in the same pool must be of the same technical specification and SKU. To enable parallel builds of the source code for different test groups, IDT for FreeRTOS copies the source code to a results folder inside the IDT for FreeRTOS extracted folder. You must reference the source code path in your build or flash command using the `testdata.sourcePath` variable. IDT for FreeRTOS replaces this variable with a temporary path of the copied source code. For more information, see IDT for FreeRTOS variables.

The following is an example `device.json` file was used to create a device pool with multiple devices.

```
[
    {
        "id": "pool-id",
        "sku": "sku",
        "features": [
           {
              "name": "Wifi",
              "value": "Yes | No"
           },
           {
              "name": "Cellular",
              "value": "Yes | No"
           },
           {
```

```
              "name": "BLE",
              "value": "Yes | No"
          },
          {
             "name": "PKCS11",
             "value": "RSA | ECC | Both"
          },
          {
              "name": "OTA",
              "value": "Yes | No",
              "configs": [
              {
                  "name": "OTADataPlaneProtocol",
                  "value": "MQTT | HTTP | None"
              }
            ]
          },
          {
             "name": "KeyProvisioning",
             "value": "Onboard | Import | Both | No"
          }
        ],
        "devices": [
          {
            "id": "device-id",
            "connectivity": {
              "protocol": "uart",
              "serialPort": "/dev/tty*"
            },
            "secureElementConfig" : {
              "publicKeyAsciiHexFilePath": "absolute-path-to/public-key-txt-file:
 contains-the-hex-bytes-public-key-extracted-from-onboard-private-key",
              "publiDeviceCertificateArn": "arn:partition:iot:region:account-
id:resourcetype:resource:qualifier",
              "secureElementSerialNumber": "secure-element-serialNo-value",
              "preProvisioned"           : "Yes | No",
              "pkcs11JITPCodeVerifyRootCertSupport": "Yes | No"
            },
            "identifiers": [
              {
                "name": "serialNo",
                "value": "serialNo-value"
              }
            ]
```

```
            }
        ]
    }
]
```

The following attributes are used in the `device.json` file:

`id`

A user-defined alphanumeric ID that uniquely identifies a pool of devices. Devices that belong to a pool must be of the same type. When a suite of tests is running, devices in the pool are used to parallelize the workload.

`sku`

An alphanumeric value that uniquely identifies the board you are testing. The SKU is used to track qualified boards.

> **Note**
>
> If you want to list your board in AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

`features`

An array that contains the device's supported features. AWS IoT Device Tester uses this information to select the qualification tests to run.

Supported values are:

`Wifi`

Indicates if your board has Wi-Fi capabilities.

`Cellular`

Indicates if your board has cellular capabilities.

`PKCS11`

Indicates the public key cryptography algorithm that the board supports. PKCS11 is required for qualification. Supported values are ECC, RSA, and `Both`. `Both` indicates the board supports both ECC and RSA.

KeyProvisioning

Indicates the method of writing a trusted X.509 client certificate onto your board.

Valid values are `Import`, `Onboard`, `Both` and `No`. `Onboard`, `Both`, or `No` key provisioning is required for qualification. `Import` alone is not a valid option for qualification.

- Use `Import` only if your board allows the import of private keys. Selecting `Import` is not a valid configuration for qualification and should be used only for testing purposes, specifically with PKCS11 test cases. `Onboard`, `Both` or `No` is required for qualification.
- Use `Onboard` if your board supports on-board private keys (for example, if your device has a secure element, or if you prefer to generate your own device key pair and certificate). Make sure you add a `secureElementConfig` element in each of the device sections and put the absolute path to the public key file in the `publicKeyAsciiHexFilePath` field.
- Use `Both` if your board supports both importing private keys and on-board key generation for key provisioning.
- Use `No` if your board doesn't support key provisioning. `No`is only a valid option when your device is also pre-provisioned.

OTA

Indicates if your board supports over-the-air (OTA) update functionality. The `OtaDataPlaneProtocol` attribute indicates which OTA dataplane protocol the device supports. OTA with either HTTP or MQTT dataplane protocol is required for qualification. To skip running OTA tests while testing, set the OTA feature to `No` and the `OtaDataPlaneProtocol` attribute to `None`. This will not be a qualification run.

BLE

Indicates if your board supports Bluetooth Low Energy (BLE).

devices.id

A user-defined unique identifier for the device being tested.

devices.connectivity.serialPort

The serial port of the host computer used to connect to the devices being tested.

devices.secureElementConfig.PublicKeyAsciiHexFilePath

Required if your board is NOT `pre-provisioned` or `PublicDeviceCertificateArn` is not provided. Since `Onboard`is a required type of Key Provisioning, this field is currently

required for the FullTransportInterfaceTLS test group. If your device is `pre-provisioned`, `PublicKeyAsciiHexFilePath` is optional and need not be included.

The following block is an absolute path to the file that contains the hex bytes public key extracted from `Onboard` private key.

```
3059 3013 0607 2a86 48ce 3d02 0106 082a
8648 ce3d 0301 0703 4200 04cd 6569 ceb8
1bb9 1e72 339f e8cf 60ef 0f9f b473 33ac
6f19 1813 6999 3fa0 c293 5fae 08f1 1ad0
41b7 345c e746 1046 228e 5a5f d787 d571
dcb2 4e8d 75b3 2586 e2cc 0c
```

If your public key is in .der format, you can hex encode the public key directly to generate the hex file.

To generate the hex file from a .der public key, enter the following **xxd** command:

```
xxd -p pubkey.der > outFile
```

If your public key is in .pem format, you can extract the base64 encoded headers and footers and decode it into binary format. Then, you hex encode the binary string to generate the hex file.

To generate a hex file for a .pem public key, do the following:

1. Run the following **base64** command to remove the base64 header and footer from the public key. The decoded key, named `base64key`, is then output to the file `pubkey.der`:

   ```
   base64 –decode base64key > pubkey.der
   ```

2. Run the following **xxd** command to convert `pubkey.der` to hex format. The resulting key is saved as *`outFile`*

   ```
   xxd -p pubkey.der > outFile
   ```

`devices.secureElementConfig.PublicDeviceCertificateArn`

The ARN of the certificate from your secure element that is uploaded to AWS IoT Core. For information about uploading your certificate to AWS IoT Core, see X.509 client certificates in the *AWS IoT Developer Guide*.

`devices.secureElementConfig.SecureElementSerialNumber`

(Optional) The serial number of the secure element. The serial number is optionally used to create device certificates for JITR key provisioning.

`devices.secureElementConfig.preProvisioned`

(Optional) Set to "Yes" if the device has a pre-provisioned secure element with locked-down credentials, that cannot import, create, or destroy objects. If this attribute is set to **Yes**, you must provide the corresponding pkcs11 labels.

`devices.secureElementConfig.pkcs11JITPCodeVerifyRootCertSupport`

(Optional) Set to **Yes** if the device's corePKCS11 implementation supports storage for JITP. This will enable the JITP `codeverify` test when testing core PKCS 11, and requires code verification key, JITP certificate, and root certificate PKCS 11 labels to be provided.

`identifiers`

(Optional) An array of arbitrary name-value pairs. You can use these values in the build and flash commands described in the next section.

Configure build, flash, and test settings

IDT for FreeRTOS builds and flashes tests on to your board automatically. To enable this, you must configure IDT to run the build and flash commands for your hardware. The build and flash command settings are configured in the `userdata.json` template file located in the `config` folder.

Configure settings for testing devices

Build, flash, and test settings are made in the `configs/userdata.json` file. The following JSON example shows how you can configure IDT for FreeRTOS to test multiple devices:

```
{
    "sourcePath": "</path/to/freertos>",
    "retainModifiedSourceDirectories": true | false,
    "freeRTOSVersion": "<freertos-version>",
```

```
    "freeRTOSTestParamConfigPath": "{{testData.sourcePath}}/path/from/source/path/to/
test_param_config.h",
    "freeRTOSTestExecutionConfigPath": "{{testData.sourcePath}}/path/from/source/path/
to/test_execution_config.h",
    "buildTool": {
        "name": "your-build-tool-name",
        "version": "your-build-tool-version",
        "command": [
            "<build command> -any-additional-flags {{testData.sourcePath}}"
        ]
    },
    "flashTool": {
        "name": "your-flash-tool-name",
        "version": "your-flash-tool-version",
        "command": [
            "<flash command> -any-additional-flags {{testData.sourcePath}} -any-
additional-flags"
        ]
    },
    "testStartDelayms": 0,
    "echoServerConfiguration": {
      "keyGenerationMethod": "EC | RSA",
      "serverPort": 9000
    },
    "otaConfiguration": {
        "otaE2EFirmwarePath": "{{testData.sourcePath}}/relative-path-to/ota-image-
generated-in-build-process",
        "otaPALCertificatePath": "/path/to/ota/pal/certificate/on/device",
        "deviceFirmwarePath" : "/path/to/firmware/image/name/on/device",
        "codeSigningConfiguration": {
            "signingMethod": "AWS | Custom",
            "signerHashingAlgorithm": "SHA1 | SHA256",
            "signerSigningAlgorithm": "RSA | ECDSA",
            "signerCertificate": "arn:partition:service:region:account-
id:resource:qualifier | /absolute-path-to/signer-certificate-file",
            "untrustedSignerCertificate": "arn:partition:service:region:account-
id:resourcetype:resource:qualifier",
            "signerCertificateFileName": "signerCertificate-file-name",
            "compileSignerCertificate": true | false,
            // ***********Use signerPlatform if you choose AWS for
 signingMethod***************
            "signerPlatform": "AmazonFreeRTOS-Default | AmazonFreeRTOS-TI-CC3220SF"

            ]
```

```
        }
    },
    **********
    This section is used for PKCS #11 labels of private key, public key, device
 certificate, code verification key, JITP certificate, and root certificate.
    When configuring PKCS11, you set up labels and you must provide the labels of the
 device certificate, public key,
    and private key for the key generation type (EC or RSA) it was created with. If
 your device supports PKCS11 storage of JITP certificate,
    code verification key, and root certificate, set
 'pkcs11JITPCodeVerifyRootCertSupport' to 'Yes' in device.json and provide the
 corresponding labels.
    **********
    "pkcs11LabelConfiguration":{
        "pkcs11LabelDevicePrivateKeyForTLS": "<device-private-key-label>",
        "pkcs11LabelDevicePublicKeyForTLS": "<device-public-key-label>",
        "pkcs11LabelDeviceCertificateForTLS": "<device-certificate-label>",
        "pkcs11LabelPreProvisionedECDevicePrivateKeyForTLS": "<preprovisioned-ec-
device-private-key-label>",
        "pkcs11LabelPreProvisionedECDevicePublicKeyForTLS": "<preprovisioned-ec-device-
public-key-label>",
        "pkcs11LabelPreProvisionedECDeviceCertificateForTLS": "<preprovisioned-ec-
device-certificate-label>",
        "pkcs11LabelPreProvisionedRSADevicePrivateKeyForTLS": "<preprovisioned-rsa-
device-private-key-label>",
        "pkcs11LabelPreProvisionedRSADevicePublicKeyForTLS": "<preprovisioned-rsa-
device-public-key-label>",
        "pkcs11LabelPreProvisionedRSADeviceCertificateForTLS": "<preprovisioned-rsa-
device-certificate-label>",
        "pkcs11LabelCodeVerifyKey": "<code-verification-key-label>",
        "pkcs11LabelJITPCertificate": "<JITP-certificate-label>",
        "pkcs11LabelRootCertificate": "<root-certificate-label>"
     }
  }
```

The following lists the attributes used in `userdata.json`:

`sourcePath`

The path to the root of the ported FreeRTOS source code.

`retainModifiedSourceDirectories`

(Optional) Checks if to retain the modified source directories used during building and flashing for debugging purposes. If set to `true`, the modified source directories are named retainedSrc and found within the results log folders in each test group run. If not included, the field defaults to `false`.

`freeRTOSTestParamConfigPath`

The path to `test_param_config.h` file for FreeRTOS-Libraries-Integration-Tests integration. This file must use the `{{testData.sourcePath}}` placeholder variable to make it relative to the source code root. AWS IoT Device Tester uses the parameters in this file to configure the tests.

`freeRTOSTestExecutionConfigPath`

The path to `test_execution_config.h` file for FreeRTOS-Libraries-Integration-Tests integration. This file must use the `{{testData.sourcePath}}` placeholder variable to make it relative to the repository root. AWS IoT Device Tester uses this file to control which tests must run.

`freeRTOSVersion`

The version of FreeRTOS including the patch version used in your implementation. See Supported versions of AWS IoT Device Tester for FreeRTOS for the FreeRTOS versions compatible with AWS IoT Device Tester for FreeRTOS.

`buildTool`

The command to build your source code. All references to the source code path in the build command must be replaced by the AWS IoT Device Tester variable `{{testData.sourcePath}}`. Use the `{{config.idtRootPath}}` placeholder to reference a build script relative to the AWS IoT Device Tester root path.

`flashTool`

The command to flash an image to your device. All references to the source code path in the flash command must be replaced by the AWS IoT Device Tester variable `{{testData.sourcePath}}`. Use the `{{config.idtRootPath}}` placeholder to reference a flash script relative to the AWS IoT Device Tester root path.

> **Note**
>
> The new integration tests structure with FRQ 2.0 doesn't require path variables such as `{{enableTests}}` and `{{buildImageName}}`. The OTA End to End tests are run with the config templates provided in the FreeRTOS-Libraries-Integration-Tests GitHub repository. If the files in the GitHub repository are present in your parent source project, the source code isn't changed between tests. If a different build image for OTA End to End is needed, you must build this image in the build script and specify it in the `userdata.json` file specified under `otaConfiguration`.

`testStartDelayms`

Specifies how many milliseconds the FreeRTOS test runner will wait before starting to run tests. This can be useful if the device under test begins to output important test information before IDT has a chance to connect and start logging due to network or other latency issues. This value is applicable to FreeRTOS test groups only, and not to other test groups that do not utilize the FreeRTOS test runner, such as the OTA tests. If you receive an error related to **expected 10 but received 5**, this field should be set to 5000.

`echoServerConfiguration`

The configuration to setup the echo server for the TLS test. This field is required.

`keyGenerationMethod`

The echo server is configured with this option. The options are EC, or RSA.

`serverPort`

The port number on which the echo server runs.

`otaConfiguration`

The configuration for OTA PAL and OTA E2E tests. This field is required.

`otaE2EFirmwarePath`

Path to the OTA bin image that IDT uses for the OTA End to End tests.

`otaPALCertificatePath`

The path to the certificate for OTA PAL test on device. This is used to verify the signature. For example, **ecdsa-sha256-signer.crt.pem**.

`deviceFirmwarePath`

The path to the hard coded name for the firmware image to boot. If your device does NOT use the file system for firmware boot, specify this field as `'NA'`. If your device uses the file system for firmware boot, specify the path or name to the firmware boot image.

`codeSigningConfiguration`

`signingMethod`

The code signing method. Possible values are AWS or Custom.

> **Note**
>
> For the Beijing and Ningxia Regions, use Custom. AWS code signing is not supported in that region.

`signerHashingAlgorithm`

The hashing algorithm supported on the device. Possible values are SHA1 or SHA256.

`signerSigningAlgorithm`

The signing algorithm supported on the device. Possible values are RSA or ECDSA.

`signerCertificate`

The trusted certificate used for OTA. For the AWS code signing method, use the Amazon Resource Name (ARN) for the trusted certificate uploaded to the AWS Certificate Manager. For the Custom code signing method, use the absolute path to the signer certificate file. For information about creating a trusted certificate, see Create a code-signing certificate.

`untrustedSignerCertificate`

The ARN or filepath for a second certificate used in some OTA tests as an untrusted certificate. For information about creating a certificate, see Create a code-signing certificate.

`signerCertificateFileName`

The file name of the code signing certificate on the device. This value must match the file name that you provided when you ran the `aws acm import-certificate` command.

`compileSignerCertificate`

Boolean value that determines the status of the signature verification certificate. Valid values are `true` and `false`.

Set this value to **true** if the code signer signature verification certificate is not provisioned or flashed. It must be compiled into the project. AWS IoT Device Tester fetches the trusted certificate and compiles it into `aws_codesigner_certificate.h`.

`signerPlatform`

The signing and hashing algorithm that AWS Code Signer uses while creating the OTA update job. Currently, the possible values for this field are `AmazonFreeRTOS-TI-CC3220SF` and `AmazonFreeRTOS-Default`.

- Choose `AmazonFreeRTOS-TI-CC3220SF` if `SHA1` and `RSA`.
- Choose `AmazonFreeRTOS-Default` if `SHA256` and `ECDSA`.
- If you need `SHA256` | `RSA` or `SHA1` | `ECDSA` for your configuration, contact us for further support.
- Configure `signCommand` if you chose `Custom` for `signingMethod`.

`signCommand`

Two placeholders `{{inputImageFilePath}}` and `{{outputSignatureFilePath}}` are required in the command. `{{inputImageFilePath}}` is the file path of the image built by IDT to be signed. `{{outputSignatureFilePath}}` is the file path of the signature which will be generated by the script.

`pkcs11LabelConfiguration`

PKCS11 label configuration requires at least one set of labels of device certificate label, public key label, and private key label to run the PKCS11 test groups. The required PKCS11 labels are based on your device configuration in the `device.json` file. If pre-provisioned is set to **Yes** in `device.json`, then the required labels must be one of the below depending on what's chosen for the PKCS11 feature.

- `PreProvisionedEC`
- `PreProvisionedRSA`

If pre-provisioned is set to **No** in `device.json`, then the required labels are:

- pkcs11LabelDevicePrivateKeyForTLS
- pkcs11LabelDevicePublicKeyForTLS
- pkcs11LabelDeviceCertificateForTLS

The following three labels are required only if you select **Yes** for pkcs11JITPCodeVerifyRootCertSupport in your device.json file.

- pkcs11LabelCodeVerifyKey
- pkcs11LabelRootCertificate
- pkcs11LabelJITPCertificate

The values for these fields should match the values defined in the FreeRTOS Porting Guide.

pkcs11LabelDevicePrivateKeyForTLS

(Optional) This label is used for the PKCS #11 label of the private key. For devices with onboard and import support of key provisioning, this label is used for testing. This label may be different than the one defined for the pre-provisioned case. If you have key provisioning set to **No** and pre-provisioned set to **Yes**, in device.json, this will be undefined.

pkcs11LabelDevicePublicKeyForTLS

(Optional) This label is used for the PKCS #11 label of the public key. For devices with onboard and import support of key provisioning, this label is used for testing. This label may be different than the one defined for pre-provisioned case. If you have key provisioning set to **No** and pre-provisioned set to **Yes**, in device.json, this will be undefined.

pkcs11LabelDeviceCertificateForTLS

(Optional) This label is used for the PKCS #11 label of the device certificate. For devices with onboard and import support of key provisioning, this label will be used for testing. This label may be different than the one defined for pre-provisioned case. If you have key provisioning set to **No** and pre-provisioned set to **Yes**, in device.json, this will be undefined.

pkcs11LabelPreProvisionedECDevicePrivateKeyForTLS

(Optional) This label is used for the PKCS #11 label of the private key. For devices with secure elements or hardware limitations, this will have a different label to preserve AWS IoT credentials. If your device supports pre-provisioning with an EC key, provide this label. When preProvisioned is set to **Yes** in device.json, this label,

`pkcs11LabelPreProvisionedRSADevicePrivateKeyForTLS`, or both must be provided. This label may be different than the one defined for onboard and import cases.

`pkcs11LabelPreProvisionedECDevicePublicKeyForTLS`

(Optional) This label is used for the PKCS #11 label of the public key. For devices with secure elements or hardware limitations, this will have a different label to preserve AWS IoT credentials. If your device supports pre-provisioning with an EC key, provide this label. When preProvisioned is set to **Yes** in `device.json`, this label, `pkcs11LabelPreProvisionedRSADevicePublicKeyForTLS`, or both must be provided. This label may be different than the one defined for onboard and import cases.

`pkcs11LabelPreProvisionedECDeviceCertificateForTLS`

(Optional) This label is used for the PKCS #11 label of the device certificate. For devices with secure elements or hardware limitations, this will have a different label to preserve AWS IoT credentials. If your device supports pre-provisioning with an EC key, provide this label. When preProvisioned is set to **Yes** in `device.json`, this label, `pkcs11LabelPreProvisionedRSADeviceCertificateForTLS`, or both must be provided. This label may be different than the one defined for onboard and import cases.

`pkcs11LabelPreProvisionedRSADevicePrivateKeyForTLS`

(Optional) This label is used for the PKCS #11 label of the private key. For devices with secure elements or hardware limitations, this will have a different label to preserve AWS IoT credentials. If your device supports pre-provisioning with an RSA key, provide this label. When preProvisioned is set to **Yes** in `device.json`, this label, `pkcs11LabelPreProvisionedECDevicePrivateKeyForTLS`, or both must be provided.

`pkcs11LabelPreProvisionedRSADevicePublicKeyForTLS`

(Optional) This label is used for the PKCS #11 label of the public key. For devices with secure elements or hardware limitations, this will have a different label to preserve AWS IoT credentials. If your device supports pre-provisioning with an RSA key, provide this label. When preProvisioned is set to **Yes** in `device.json`, this label, `pkcs11LabelPreProvisionedECDevicePublicKeyForTLS`, or both must be provided.

`pkcs11LabelPreProvisionedRSADeviceCertificateForTLS`

(Optional) This label is used for the PKCS #11 label of the device certificate. For devices with secure elements or hardware limitations, this will have a different label to preserve AWS IoT credentials. If your device supports pre-provisioning with an RSA

key, provide this label. When preProvisioned is set to **Yes** in `device.json`, this label, `pkcs11LabelPreProvisionedECDeviceCertificateForTLS`, or both must be provided.

`pkcs11LabelCodeVerifyKey`

(Optional) This label is used for the PKCS #11 label of the code verification key. If your device has PKCS #11 storage support of the JITP certificate, code verification key, and root certificate, provide this label. When `pkcs11JITPCodeVerifyRootCertSupport` in `device.json` is set to **Yes**, this label must be provided.

`pkcs11LabelJITPCertificate`

(Optional) This label is used for the PKCS #11 label of the JITP certificate. If your device has PKCS #11 storage support of the JITP certificate, code verification key, and root certificate, provide this label. When `pkcs11JITPCodeVerifyRootCertSupport` in `device.json` is set to **Yes**, this label must be provided.

IDT for FreeRTOS variables

The commands to build your code and flash the device might require connectivity or other information about your devices to run successfully. AWS IoT Device Tester allows you to reference device information in flash and build commands using JsonPath. By using simple JsonPath expressions, you can fetch the required information specified in your `device.json` file.

Path variables

IDT for FreeRTOS defines the following path variables that can be used in command lines and configuration files:

`{{testData.sourcePath}}`

Expands to the source code path. If you use this variable, it must be used in both the flash and build commands.

`{{device.connectivity.serialPort}}`

Expands to the serial port.

`{{device.identifiers[?(@.name == 'serialNo')].value[0]}}`

Expands to the serial number of your device.

`{{config.idtRootPath}}`

Expands to the AWS IoT Device Tester root path.

UI for IDT for FreeRTOS qualification suite 2.0 (FRQ 2.0)

AWS IoT Device Tester for FreeRTOS (IDT for FreeRTOS) includes a web-based user interface (UI) where you can interact with the IDT command line application and related configuration files. You use the IDT for FreeRTOS UI to create a new configuration, or modify an existing one, for your device. You can also use the UI to call the IDT application and run the FreeRTOS tests against your device.

For information about how to use the command line to run qualification tests, see First test of your microcontroller board.

This section describes the prerequisites for the IDT for FreeRTOS UI and how to run qualifications tests from the UI.

Topics

- Set up the IDT prerequisites
- Configure AWS credentials to use the IDT UI
- Open the IDT for FreeRTOS UI
- Create a new configuration
- Modify an existing configuration
- Run qualification tests

Set up the IDT prerequisites

To run tests through the AWS IoT Device Tester (IDT) for FreeRTOS UI, you must complete the prerequisites on the Set up the LTS qualification prerequisites page for IDT FreeRTOS Qualification (FRQ) 2.x.

Configure AWS credentials to use the IDT UI

You must configure your IAM user credentials for the AWS user you created in Create an AWS account. You can specify your credentials in one of two ways:

- In a credentials file
- As environment variables

Configure AWS credentials with a credentials file

IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

The location of the credentials file varies based on the operating system you use:

- **macOS and Linux** – `~/.aws/credentials`
- **Windows** – `C:\Users\UserName\.aws\credentials`

Add your AWS credentials to the `credentials` file in the following format:

```
[default]
aws_access_key_id = your_access_key_id
aws_secret_access_key = your_secret_access_key
```

> **Note**
>
> If you don't use the `default` AWS profile, you must specify the profile name in the IDT for FreeRTOS UI. For more information about profiles, see Configuration and credential file settings.

Configure AWS credentials with environment variables

Environment variables are variables maintained by the operating system and used by system commands. They're not saved if you close the SSH session. The IDT for FreeRTOS UI uses the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to store your AWS credentials.

To set these variables on Linux, macOS, or Unix, use **export**:

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
```

To set these variables on Windows, use **set**:

```
set AWS_ACCESS_KEY_ID=your_access_key_id
set AWS_SECRET_ACCESS_KEY=your_secret_access_key
```

Open the IDT for FreeRTOS UI

This topic explains how to open the IDT for FreeRTOS UI to use the FreeRTOS qualification suite.

To open the IDT for FreeRTOS UI

1. Download a supported IDT for FreeRTOS version. Then extract the downloaded archive to a directory that you have read and write permissions for.
2. Navigate to the IDT for FreeRTOS installation directory:

   ```
   cd devicetester-extract-location/bin
   ```
3. Run the following command to open the IDT for FreeRTOS UI:

   Linux

   ```
   .devicetester_ui_linux_x86-64
   ```

   Windows

   ```
   ./devicetester_ui_win_x64-64
   ```

   macOS

   ```
   ./devicetester_ui_mac_x86-64
   ```

   > **Note**
   >
   > In macOS, to allow your system to run the UI, go to **System Preferences -> Security & Privacy**. When you run the tests, you may need to do this three more times. this

   The IDT for FreeRTOS UI opens in your default browser. The latest three major versions of the following browsers support the UI:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge
- Apple Safari for macOS

> **Note**
>
> For a better experience, we recommend Google Chrome or Mozilla Firefox to access the IDT for FreeRTOS UI. Microsoft Internet Explorer isn't supported by the UI.

> **Important**
>
> You must configure your AWS credentials before you open the UI. If you haven't configured your credentials, close the IDT for FreeRTOS UI browser window, follow the steps in Configure AWS credentials to use the IDT UI, and then reopen the IDT for FreeRTOS UI.

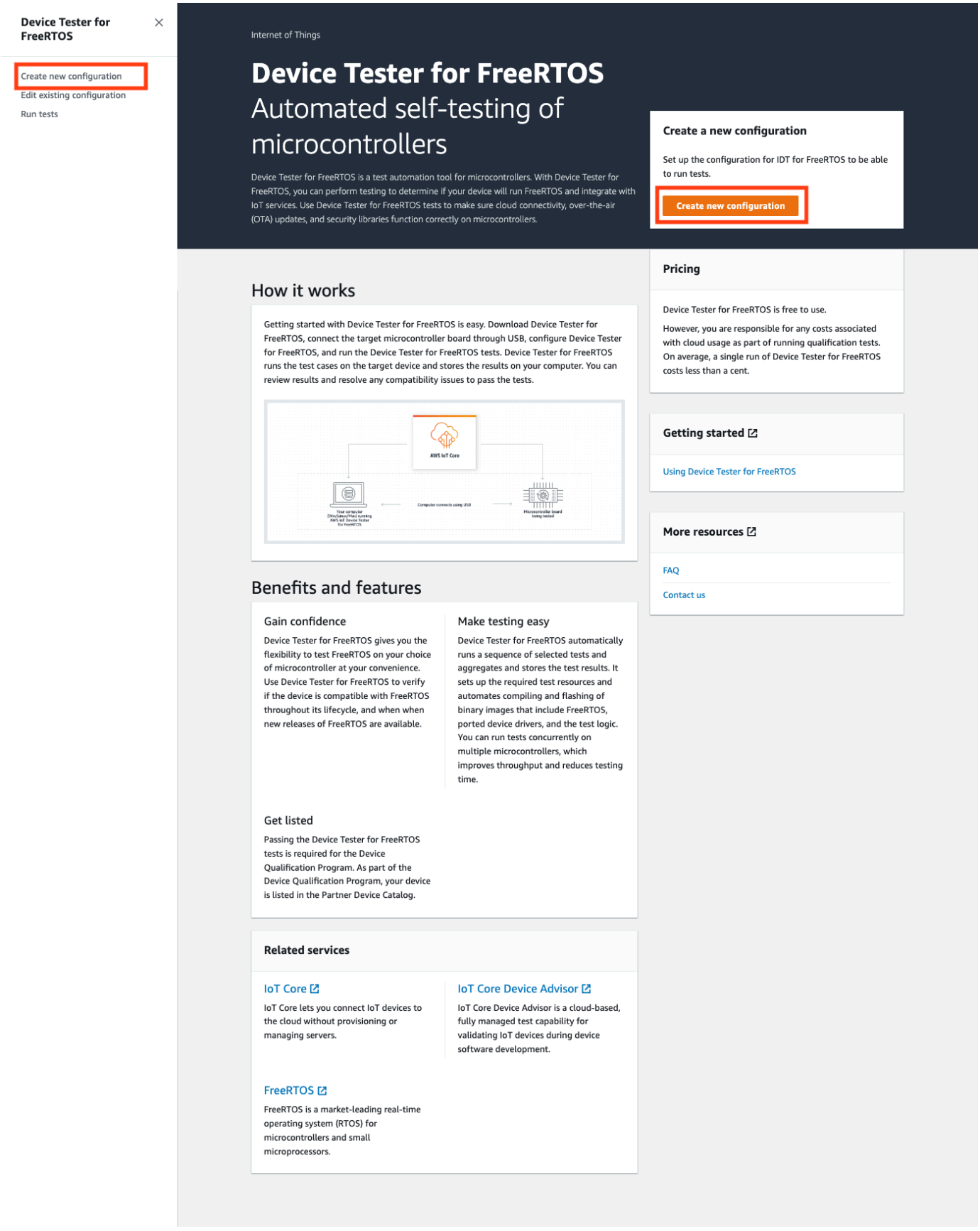
Create a new configuration

If you're a first-time user, you must create a new configuration to set up the JSON configuration files that IDT for FreeRTOS requires to run tests. You can then run tests or modify the created configuration.

For examples of the `config.json`, `device.json`, and `userdata.json` files, see First test of your microcontroller board.

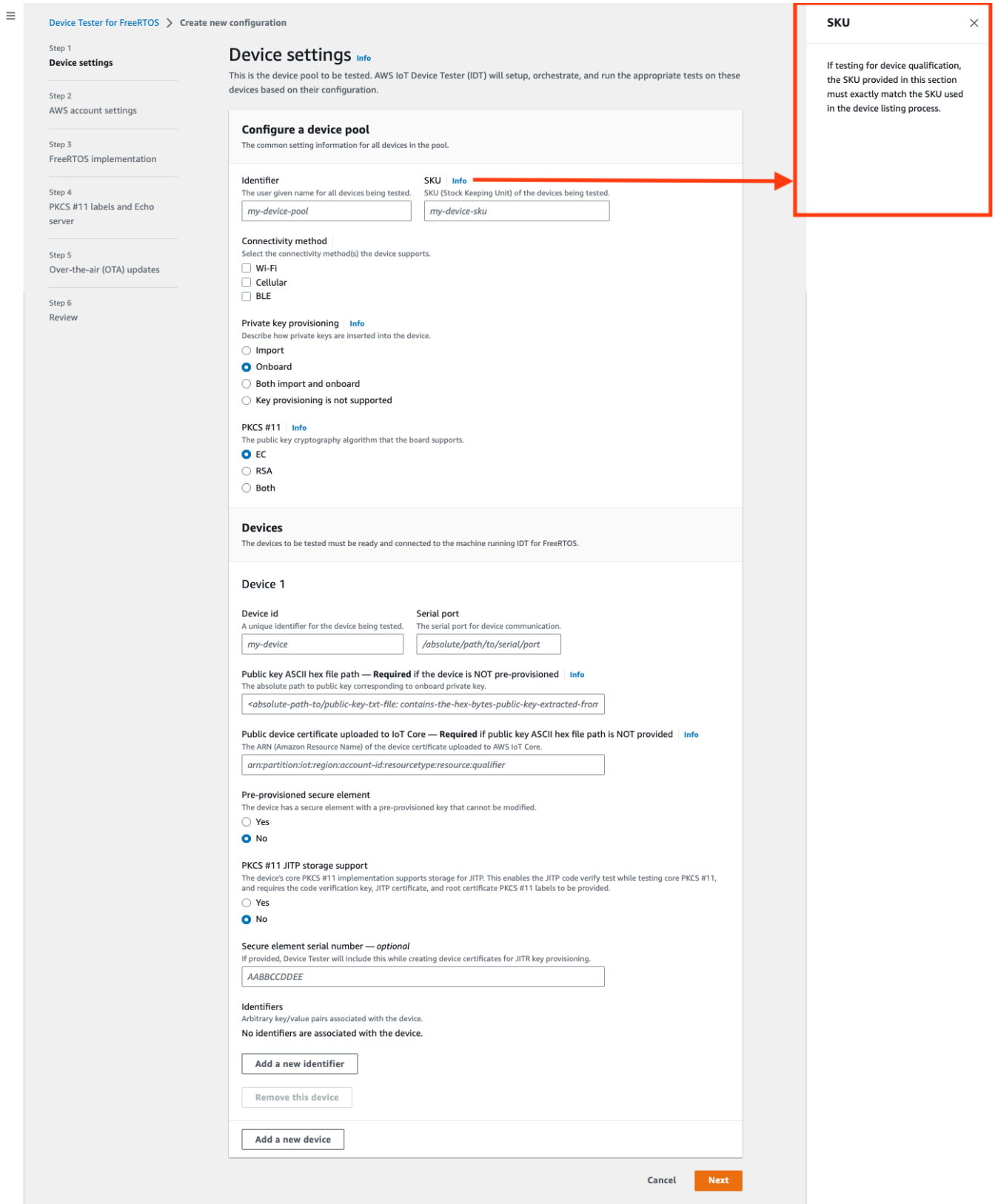
To create a new configuration

1. In the IDT for FreeRTOS UI, open the navigation menu, and choose **Create new configuration**.

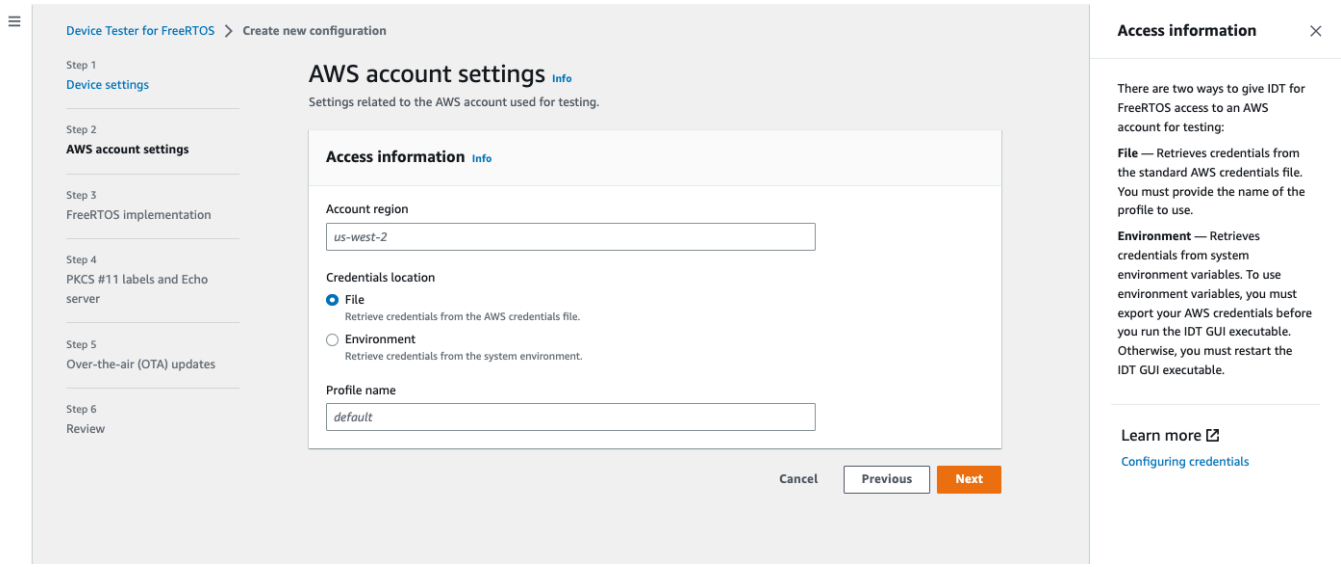

Device Tester for FreeRTOS
Create new configuration
Edit existing configuration
Run tests
Internet of Things
Device Tester for FreeRTOS
Automated self-testing of microcontrollers
Device Tester for FreeRTOS is a test automation tool for microcontrollers. With Device Tester for FreeRTOS, you can perform testing to determine if your device will run FreeRTOS and integrate with IoT services. Use Device Tester for FreeRTOS tests to make sure cloud connectivity, over-the-air (OTA) updates, and security libraries function correctly on microcontrollers.
Create a new configuration
Set up the configuration for IDT for FreeRTOS to be able to run tests.
Create new configuration
How it works
Getting started with Device Tester for FreeRTOS is easy. Download Device Tester for FreeRTOS, connect the target microcontroller board through USB, configure Device Tester for FreeRTOS, and run the Device Tester for FreeRTOS tests. Device Tester for FreeRTOS runs the test cases on the target device and stores the results on your computer. You can review results and resolve any compatibility issues to pass the tests.
AWS IoT Core
Computer connects using USB
Benefits and features
Gain confidence
Device Tester for FreeRTOS gives you the flexibility to test FreeRTOS on your choice of microcontroller at your convenience. Use Device Tester for FreeRTOS to verify if the device is compatible with FreeRTOS throughout its lifecycle, and when when new releases of FreeRTOS are available.
Make testing easy
Device Tester for FreeRTOS automatically runs a sequence of selected tests and aggregates and stores the test results. It sets up the required test resources and automates compiling and flashing of binary images that include FreeRTOS, ported device drivers, and the test logic. You can run tests concurrently on multiple microcontrollers, which improves throughput and reduces testing time.
Get listed
Passing the Device Tester for FreeRTOS tests is required for the Device Qualification Program. As part of the Device Qualification Program, your device is listed in the Partner Device Catalog.
Related services
IoT Core
IoT Core lets you connect IoT devices to the cloud without provisioning or managing servers.
IoT Core Device Advisor
IoT Core Device Advisor is a cloud-based, fully managed test capability for validating IoT devices during device software development.
FreeRTOS
FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors.
Pricing
Device Tester for FreeRTOS is free to use.
However, you are responsible for any costs associated with cloud usage as part of running qualification tests. On average, a single run of Device Tester for FreeRTOS costs less than a cent.
Getting started
Using Device Tester for FreeRTOS
More resources
FAQ
Contact us

2. Follow the configuration wizard to enter the IDT configuration settings used to run qualification tests. The wizard configures the following settings in JSON configuration files located in the *devicetester-extract-location*/config directory.

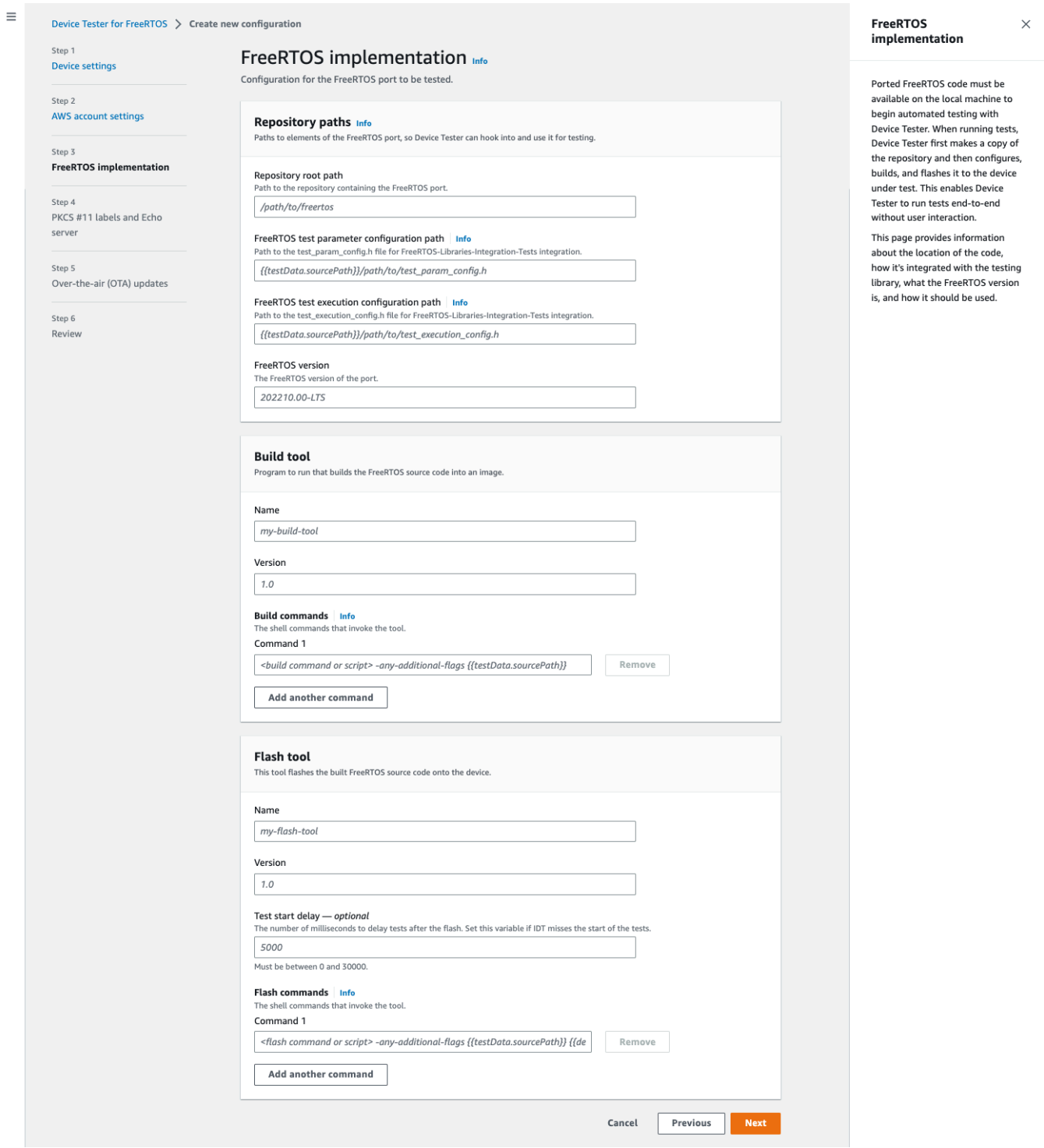
   - **Device settings** – The device pool settings for the devices to be tested. These settings are configured in the `id` and `sku` fields, and the **devices** block for the device pool in the `config.json` file.



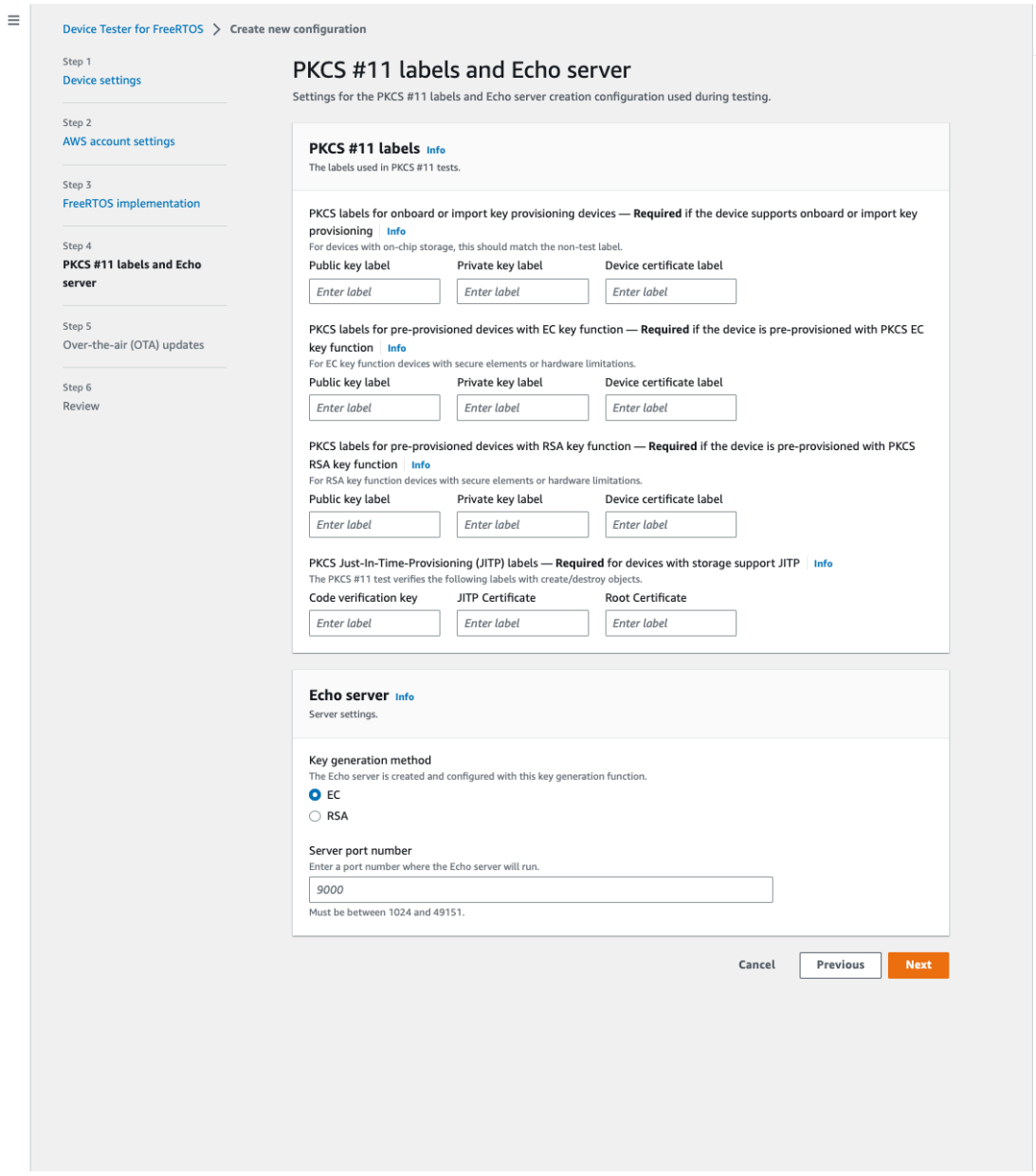
- **AWS account settings** – The AWS account information that IDT for FreeRTOS uses to create AWS resources during test runs. These settings are configured in the `config.json` file.



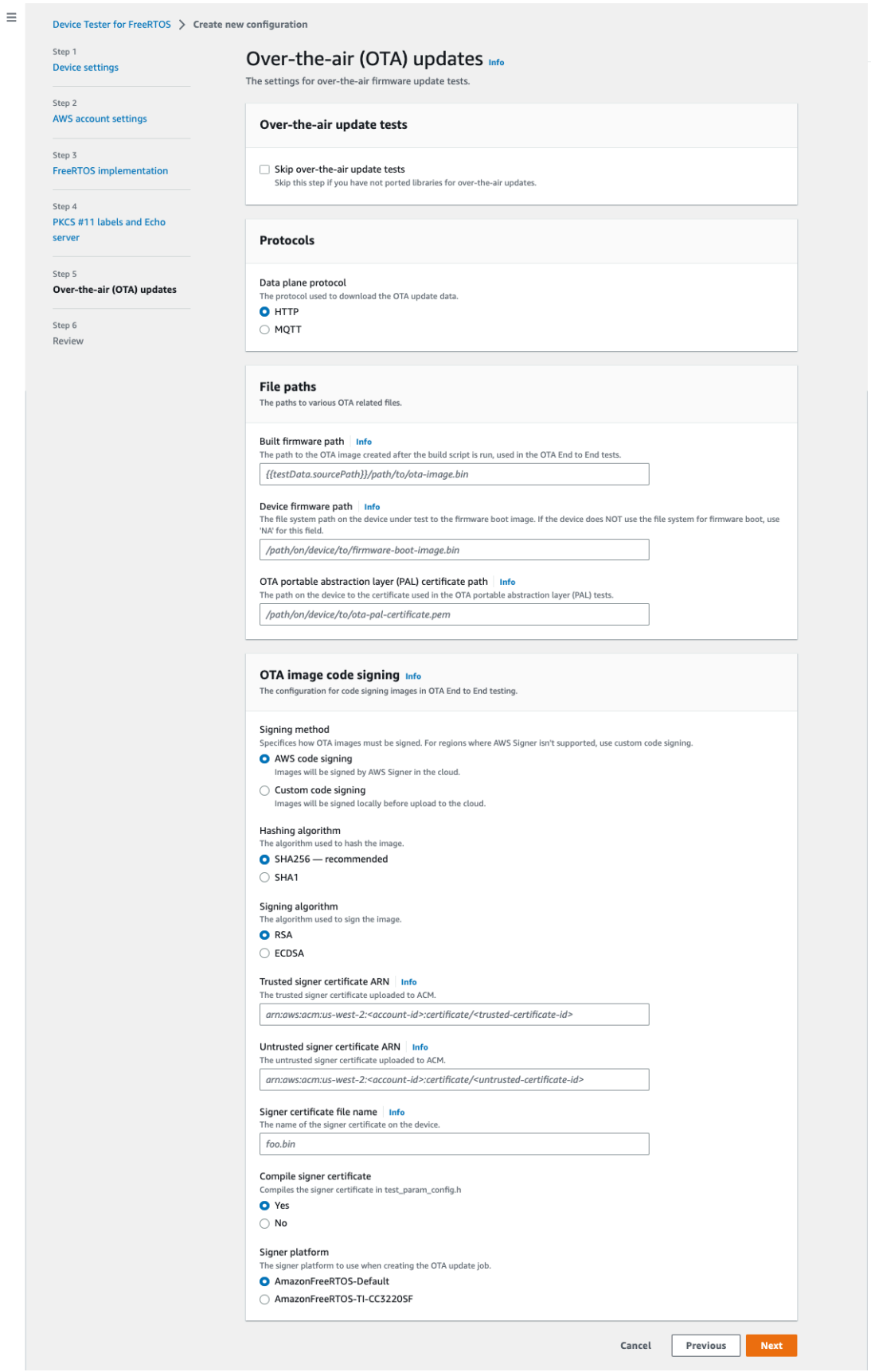
- **FreeRTOS implementation** – The absolute path to the FreeRTOS repository and ported code, and the FreeRTOS version you want to run IDT FRQ on. The paths to the execution and parameter config header files from the `FreeRTOS-Libraries-Integration-Tests` GitHub repository. The build and flash commands for your hardware that allow IDT to build and flash tests onto your board automatically. These settings are configured in the `userdata.json` file.



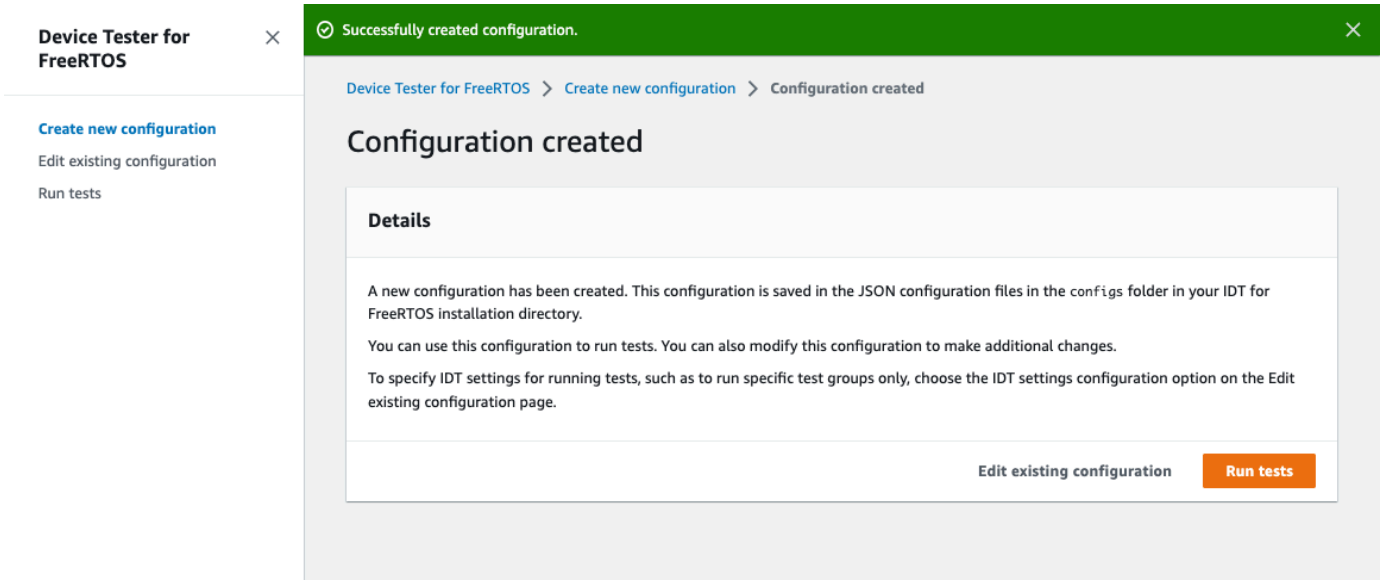
- **PKCS #11 labels and Echo server** – The PKCS #11 labels that correspond to the keys provisioned in your hardware based on the key functionality and key provisioning method. The echo server configuration settings for the Transport Interface tests. These settings are configured in the `userdata.json` and `device.json` files.



- **Over-the-air (OTA) updates** – The settings that control OTA functionality tests. These settings are configured in the `features` block of the `device.json` and `userdata.json` files.


Device Tester for FreeRTOS > Create new configuration
Step 1
Device settings
Step 2
AWS account settings
Step 3
FreeRTOS implementation
Step 4
PKCS #11 labels and Echo server
Step 5
Over-the-air (OTA) updates
Step 6
Review
Over-the-air (OTA) updates Info
The settings for over-the-air firmware update tests.
Over-the-air update tests
Skip over-the-air update tests
Skip this step if you have not ported libraries for over-the-air updates.
Protocols
Data plane protocol
The protocol used to download the OTA update data.
HTTP
MQTT
File paths
The paths to various OTA related files.
Built firmware path Info
The path to the OTA image created after the build script is run, used in the OTA End to End tests.
{{testData.sourcePath}}/path/to/ota-image.bin
Device firmware path Info
The file system path on the device under test to the firmware boot image. If the device does NOT use the file system for firmware boot, use 'NA' for this field.
/path/on/device/to/firmware-boot-image.bin
OTA portable abstraction layer (PAL) certificate path Info
The path on the device to the certificate used in the OTA portable abstraction layer (PAL) tests.
/path/on/device/to/ota-pal-certificate.pem
OTA image code signing Info
The configuration for code signing images in OTA End to End testing.
Signing method
Specifices how OTA images must be signed. For regions where AWS Signer isn't supported, use custom code signing.
AWS code signing
Images will be signed by AWS Signer in the cloud.
Custom code signing
Images will be signed locally before upload to the cloud.
Hashing algorithm
The algorithm used to hash the image.
SHA256 — recommended
SHA1
Signing algorithm
The algorithm used to sign the image.
RSA
ECDSA
Trusted signer certificate ARN Info
The trusted signer certificate uploaded to ACM.
arn:aws:acm:us-west-2:<account-id>:certificate/<trusted-certificate-id>
Untrusted signer certificate ARN Info
The untrusted signer certificate uploaded to ACM.
arn:aws:acm:us-west-2:<account-id>:certificate/<untrusted-certificate-id>
Signer certificate file name Info
The name of the signer certificate on the device.
foo.bin
Compile signer certificate
Compiles the signer certificate in test_param_config.h
Yes
No
Signer platform
The signer platform to use when creating the OTA update job.
AmazonFreeRTOS-Default
AmazonFreeRTOS-TI-CC3220SF
Cancel
Previous
Next
Over-the-air (OTA) updates
IDT for FreeRTOS runs tests to verify OTA update behavior, including end-to-end (E2E) and portable abstraction layer (PAL) tests.
These tests are required to qualify a device.
Learn more
FreeRTOS OTA Update tests

3. On the **Review** page, verify your configuration information.



After you finish reviewing your configuration, to run your qualification tests, choose **Run tests**.

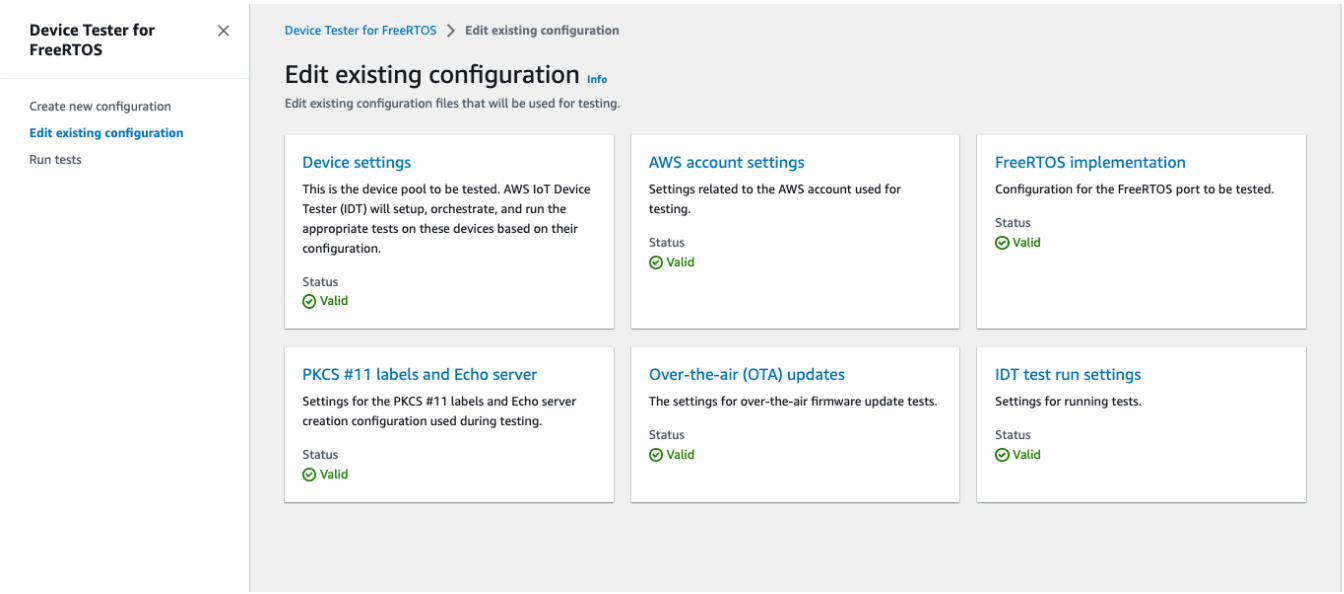
Modify an existing configuration

If you have already set up configuration files for IDT for FreeRTOS, you can use the IDT for FreeRTOS UI to modify your existing configuration. The existing configuration files must be located in the *devicetester-extract-location*`/config` directory.

To modify a configuration

1. In the IDT for FreeRTOS UI, open the navigation menu, and choose **Edit existing configuration**.

   The configuration dashboard displays information about your existing configuration settings. If a configuration is incorrect or unavailable, the status for that configuration is `Error validating configuration`.

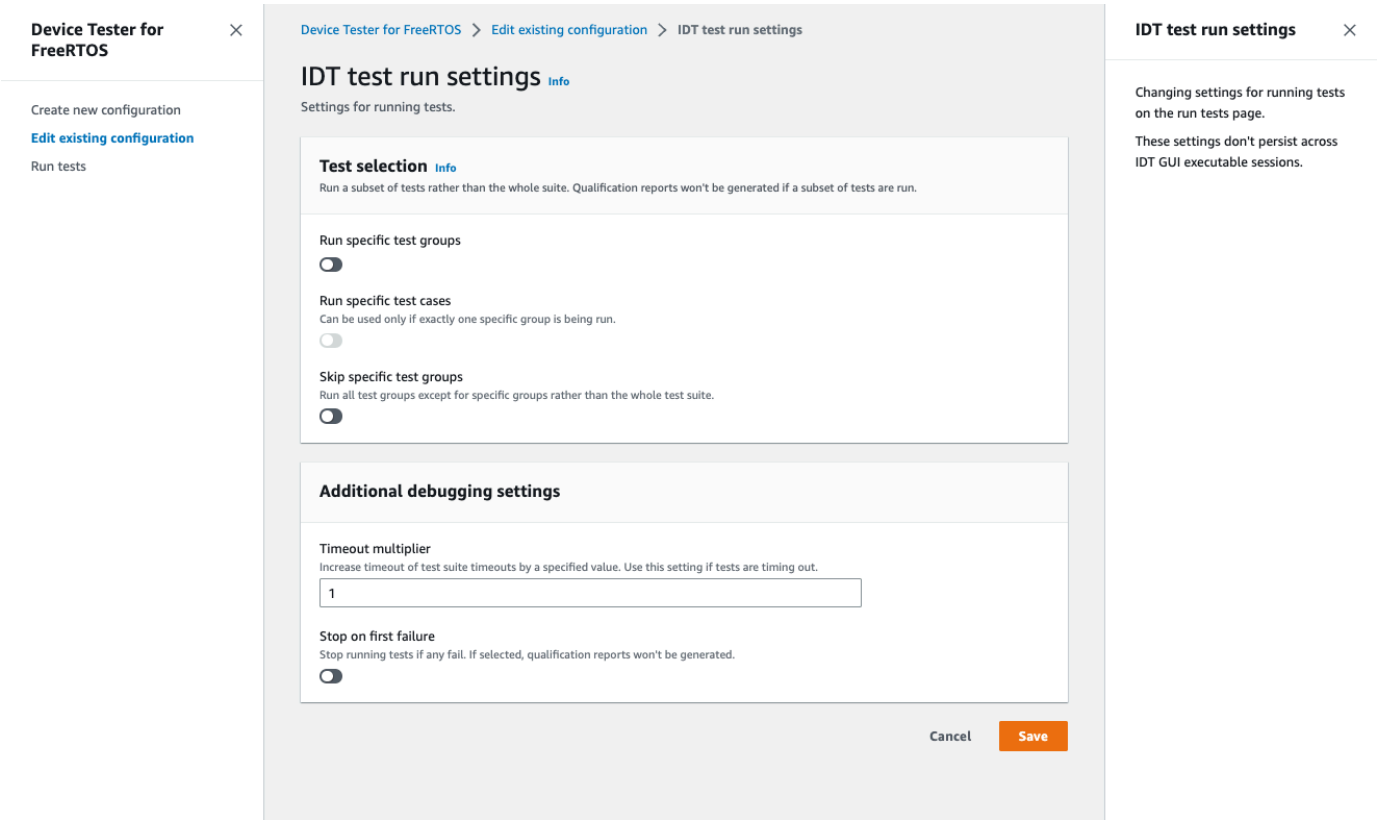


2. To modify an existing configuration setting, complete the following steps:

   a. Choose the name of a configuration setting to open its settings page.

   b. Modify the settings, and then choose **Save** to regenerate the corresponding configuration file.

3. To modify the IDT for FreeRTOS test run settings, choose **IDT test run settings** in the edit view:



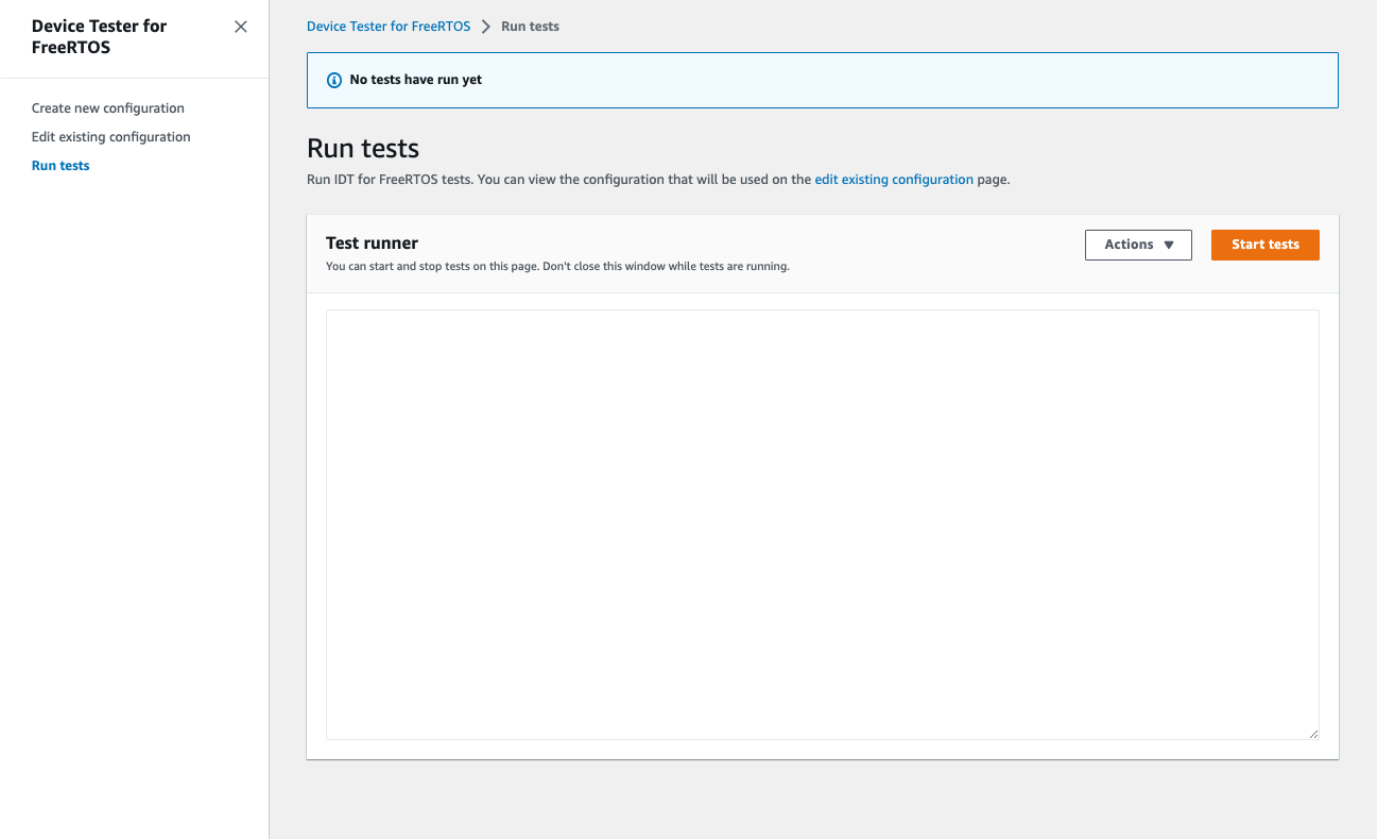
After you finish modifying your configuration, verify that all of your configuration settings pass validation. If the status for each configuration setting is `Valid`, you can run your qualification tests with this configuration.

Run qualification tests

After you create a configuration for the IDT for FreeRTOS UI you can run your qualification tests.

To run qualification tests

1. In the navigation menu, choose **Run tests**.
2. Choose **Start tests** to start the test run. By default, all applicable tests are run for your device configuration. IDT for FreeRTOS generates a qualification report when all tests finish.

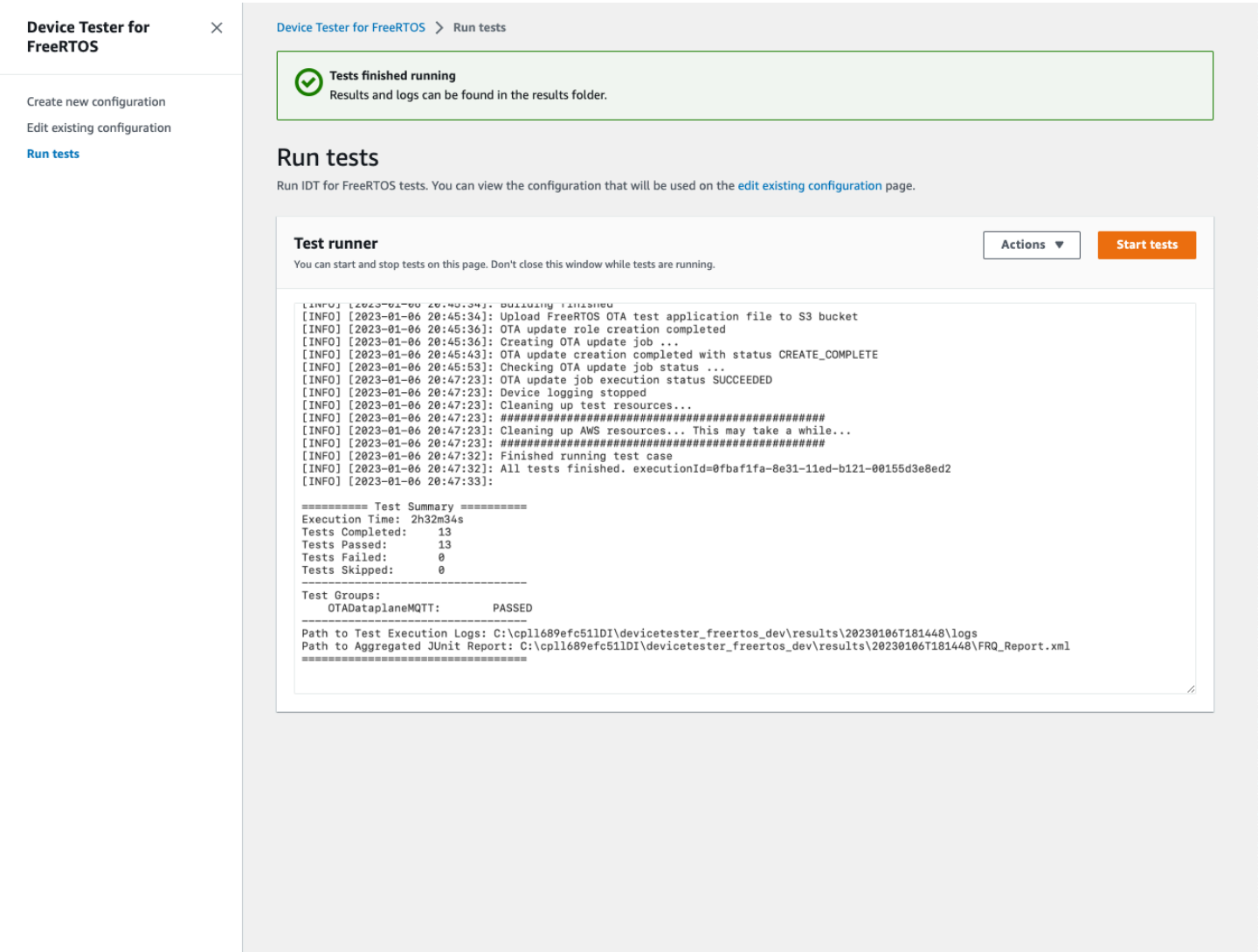


IDT for FreeRTOS runs the qualification tests. It then displays the test run summary and any errors in the **Test runner** console. After the test run is complete, you can view the test results and logs from the following locations:

- Test results are located in the *devicetester-extract-location*`/results/`*execution-id* directory.

- Test logs are located in the *devicetester-extract-location*/results/*execution-id*/logs directory.

For more information about test results and logs, see View the IDT for FreeRTOSresults and View the IDT for FreeRTOSlogs.



Run the FreeRTOS qualification 2.0 suite

Use the AWS IoT Device Tester for FreeRTOS executable to interact with IDT for FreeRTOS. The following command line examples show you how to run the qualification tests for a device pool (a set of identical devices).

IDT v4.5.2 and later

```
devicetester_[linux | mac | win] run-suite  \
    --suite-id suite-id  \
    --group-id group-id  \
    --pool-id your-device-pool \
    --test-id test-id  \
    --userdata userdata.json
```

Runs a suite of tests on a pool of devices. The `userdata.json` file must be located in the *devicetester_extract_location*/devicetester_freertos_*[win|mac|linux]*/`configs/` directory.

> **Note**
>
> If you're running IDT for FreeRTOS on Windows, use forward slashes (/) to specify the path to the `userdata.json` file.

Use the following command to run a specific test group:

```
devicetester_[linux | mac | win] run-suite  \
    --suite-id FRQ_1.99.0  \
    --group-id group-id  \
    --pool-id pool-id  \
    --userdata userdata.json
```

The `suite-id` and `pool-id` parameters are optional if you're running a single test suite on a single device pool (that is, you have only one device pool defined in your `device.json` file).

Use the following command to run a specific test case in a test group:

```
devicetester_[linux | mac | win_x86-64] run-suite  \
    --group-id group-id  \
    --test-id test-id
```

You can use the `list-test-cases` command to list the test cases in a test group.

IDT for FreeRTOS command line options

group-id

(Optional) The test groups to run, as a comma-separated list. If not specified, IDT runs all test groups in the test suite.

pool-id

(Optional) The device pool to test. This is required if you define multiple device pools in `device.json`. If you only have one device pool, you can omit this option.

suite-id

(Optional) The test suite version to run. If not specified, IDT uses the latest version in the tests directory on your system.

test-id

(Optional) The tests to run, as a comma-separated list. If specified, `group-id` must specify a single group.

Example

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id FreeRTOSVersion --
test-id FreeRTOSVersion
```

h

Use the help option to learn more about `run-suite` options.

Example

Example

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

IDT for FreeRTOS commands

The IDT for FreeRTOS command supports the following operations:

IDT v4.5.2 and later

help

Lists information about the specified command.

`list-groups`

Lists the groups in a given suite.

`list-suites`

Lists the available suites.

`list-supported-products`

Lists the supported products and test suite versions.

`list-supported-versions`

Lists the FreeRTOS and test suite versions supported by the current IDT version.

`list-test-cases`

Lists the test cases in a specified group.

`run-suite`

Runs a suite of tests on a pool of devices.

Use the `--suite-id` option to specify a test suite version, or omit it to use the latest version on your system.

Use the `--test-id` to run an individual test case.

Example

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id FreeRTOSVersion --
test-id FreeRTOSVersion
```

> **Note**
>
> Starting in IDT v3.0.0, IDT checks online for newer test suites. For more information, see Test suite versions.

View the IDT for FreeRTOSresults

While running, IDT writes errors to the console, log files, and test reports. After IDT completes the qualification test suite, it writes a test run summary to the console and generates two test reports.

These reports can be found in *devicetester-extract-location*/results/*execution-id*/. Both reports capture the results from the qualification test suite execution.

The awsiotdevicetester_report.xml is the qualification test report that you submit to AWS to list your device in the AWS Partner Device Catalog. The report contains the following elements:

- The IDT for FreeRTOS version.
- The FreeRTOS version that was tested.
- The features of FreeRTOS that are supported by the device based on the tests passed.
- The SKU and the device name specified in the device.json file.
- The features of the device specified in the device.json file.
- The aggregate summary of test case results.
- A breakdown of test case results by libraries that were tested based on the device features.

The FRQ_Report.xml is a report in standard JUnit XML format. You can integrate it into CI/CD platforms like Jenkins, Bamboo, and so on. The report contains the following elements:

- An aggregate summary of test case results.
- A breakdown of test case results by libraries that were tested based on the device features.

Interpret the IDT for FreeRTOS results

The report section in awsiotdevicetester_report.xml or FRQ_Report.xml lists the results of the tests that are executed.

The first XML tag <testsuites> contains the overall summary of the test execution. For example:

```
<testsuites name="FRQ results" time="5633" tests="184" failures="0"
errors="0" disabled="0">
```

Attributes used in the <testsuites> tag

name

The name of the test suite.

time

The time, in seconds, it took to run the qualification suite.

`tests`

The number of test cases executed.

`failures`

The number of test cases that were run, but did not pass.

`errors`

The number of test cases that IDT for FreeRTOS couldn't execute.

`disabled`

This attribute is not used and can be ignored.

If there are no test case failures or errors, your device meets the technical requirements to run FreeRTOS and can interoperate with AWS IoT services. If you choose to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the event of test case failures or errors, you can identify the test case that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag shows the test case result summary for a test group.

```
<testsuite name="FreeRTOSVersion" package="" tests="1" failures="0"
time="2" disabled="0" errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but with an attribute called `skipped` that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each of the test cases that were executed for a test group. For example:

```
<testcase classname="FRQ FreeRTOSVersion" name="FreeRTOSVersion"
attempts="1"></testcase>
```

Attributes used in the `<awsproduct>` tag

`name`

The name of the product being tested.

`version`

The version of the product being tested.

`features`

The features validated. Features marked as `required` are required to submit your board for qualification. The following snippet shows how this appears in the `awsiotdevicetester_report.xml` file.

```
<feature name="core-freertos" value="not-supported" type="required"></feature>
```

Features marked as `optional` are not required for qualification. The following snippets show optional features.

```
<feature name="ota-dataplane-mqtt" value="not-supported" type="optional"></feature>
  <feature name="ota-dataplane-http" value="not-supported" type="optional"></
feature>
```

If there are no test failures or errors for the required features, your device meets the technical requirements to run FreeRTOS and can interoperate with AWS IoT services. If you want to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the event of test failures or errors, you can identify the test that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag show the test result summary for a test group. For example:

```
<testsuite name="FreeRTOSVersion" package="" tests="1" failures="1" time="2"
 disabled="0" errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but has a `skipped` attribute that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each executed test for a test group. For example:

```
<testcase classname="FreeRTOSVersion" name="FreeRTOSVersion"></testcase>
```

Attributes used in the `<testcase>` tag

`name`

The name of the test case.

`attempts`

The number of times IDT for FreeRTOS executed the test case.

When a test fails or an error occurs, `<failure>` or `<error>` tags are added to the `<testcase>` tag with information for troubleshooting. For example:

```
<testcase classname="FRQ FreeRTOSVersion" name="FreeRTOSVersion">
      <failure type="Failure">Reason for the test case failure</failure>
      <error>Reason for the test case execution error</error>
  </testcase>
```

For more information, see Troubleshoot errors.

View the IDT for FreeRTOSlogs

You can find logs that IDT for FreeRTOS generates from test execution in *devicetester-extract-location*/results/*execution-id*/logs. Two sets of logs are generated:

- `test_manager.log`

  Contains logs generated from IDT for FreeRTOS (for example, logs related configuration and report generation).

- *test_group_id*/*test_case_id*/*test_case_id*.log

  The log file for a test case, including output from the device under test. The log file is named according to the test group and test case that was run.

IDT with FreeRTOS qualification suite 1.0 (FRQ 1.0)

Important

As of October 2022, AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 does not generate signed qualification reports. You cannot qualify new AWS IoT FreeRTOS devices to list in the AWS Partner Device Catalog through the AWS Device Qualification Program using IDT FRQ 1.0 versions. While you can't qualify FreeRTOS devices using IDT FRQ 1.0, you can continue to test your FreeRTOS devices with FRQ 1.0. We recommend

that you use IDT FRQ 2.0 to qualify and list FreeRTOS devices in the AWS Partner Device Catalog.

You can use IDT for FreeRTOS qualification to verify that the FreeRTOS operating system works locally on your device and can communicate with AWS IoT. Specifically, it verifies that the porting layer interfaces for the FreeRTOS libraries are implemented correctly. It also performs end-to-end tests with AWS IoT Core. For example, it verifies your board can send and receive MQTT messages and process them correctly. The tests run by IDT for FreeRTOS are defined in the FreeRTOS GitHub repository.

The tests run as embedded applications that are flashed onto your board. The application binary images include FreeRTOS, the semiconductor vendor's ported FreeRTOS interfaces, and board device drivers. The purpose of the tests is to verify the ported FreeRTOS interfaces function correctly on top of the device drivers.

IDT for FreeRTOS generates test reports that you can submit to AWS IoT to add your hardware to the AWS Partner Device Catalog. For more information, see AWS Device Qualification Program.

IDT for FreeRTOS runs on a host computer (Windows, macOS, or Linux) that is connected to the board to be tested. IDT executes test cases and aggregates results. It also provides a command line interface to manage test execution.

In addition to testing devices, IDT for FreeRTOS creates resources (for example, AWS IoT things, FreeRTOS groups, Lambda functions, and so on) to facilitate the qualification process. To create these resources, IDT for FreeRTOS uses the AWS credentials configured in the `config.json` to make API calls on your behalf. These resources are provisioned at various times during a test.

When you run IDT for FreeRTOS on your host computer, it performs the following steps:

1. Loads and validates your device and credentials configuration.
2. Performs selected tests with the required local and cloud resources.
3. Cleans up local and cloud resources.
4. Generates tests reports that indicate if your board passed the tests required for qualification.

Topics

- Setup the 1.0 qualification prerequisites

- First test of your microcontroller board
- Use the IDT user interface to run the FreeRTOS qualification suite
- Run Bluetooth Low Energy tests
- Run the FreeRTOS qualification suite
- View the IDT for FreeRTOS results
- Interpret the IDT for FreeRTOS results
- View the IDT for FreeRTOS logs

Setup the 1.0 qualification prerequisites

This section describes the prerequisites for testing microcontrollers with AWS IoT Device Tester.

Download FreeRTOS

You can download a release of FreeRTOS from GitHub with the following command:

```
git clone --branch <FREERTOS_RELEASE_VERSION> --recurse-submodules https://github.com/
aws/amazon-freertos.git
cd amazon-freertos
git submodule update --checkout --init --recursive
```

where <FREERTOS_RELEASE_VERSION> is a version of FreeRTOS (for example, 202007.00) corresponding to an IDT version listed in Supported versions of AWS IoT Device Tester. This ensures you have the full source code, including submodules, and are using the correct version of IDT for your version of FreeRTOS, and vice versa.

Windows has a path length limitation of 260 characters. The path structure of FreeRTOS is many levels deep, so if you are using Windows, keep your file paths under the 260-character limit. For example, clone FreeRTOS to `C:\FreeRTOS` rather than `C:\Users\username\programs\projects\myproj\FreeRTOS\`.

FreeRTOS qualification with LTS libraries

- In order for your microcontroller to be designated as supporting long-term support (LTS) based versions of FreeRTOS in the AWS Partner Device Catalog, you must provide a manifest file. For more information, see the FreeRTOS Qualification Checklist in the *FreeRTOS Qualification Guide*.

- In order to validate that your microcontroller supports LTS based versions of FreeRTOS and qualify it for submission to the AWS Partner Device Catalog, you must use AWS IoT Device Tester (IDT) with FreeRTOS Qualification (FRQ) test suite version v1.4.x.
- Support for LTS based versions of FreeRTOS is limited to the 202012.xx version of FreeRTOS.

Download IDT for FreeRTOS

Every version of FreeRTOS has a corresponding version of IDT for FreeRTOS to perform qualification tests. Download the appropriate version of IDT for FreeRTOS from Supported versions of AWS IoT Device Tester.

Extract IDT for FreeRTOS to a location on the file system where you have read and write permissions. Because Microsoft Windows has a character limit for the path length, extract IDT for FreeRTOS into a root directory such as `C:\` or `D:\`.

> **Note**
>
> We don't recommend that multiple users run IDT from a shared location, such as an NFS directory or a Windows network shared folder. This may result in crashes or data corruption. We recommend that you extract the IDT package to a local drive.

Create and configure an AWS account

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign

administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

AWS IoT Device Tester managed policy

The `AWSIoTDeviceTesterForFreeRTOSFullAccess` managed policy contains the following AWS IoT Device Tester permissions for version checking, auto update features, and collection of metrics.

- `iot-device-tester:SupportedVersion`

  Grants AWS IoT Device Tester permission to fetch the list of supported products, test suites and IDT versions.
- `iot-device-tester:LatestIdt`

  Grants AWS IoT Device Tester permission to fetch the latest IDT version available for download.
- `iot-device-tester:CheckVersion`

  Grants AWS IoT Device Tester permission to check version compatibility for IDT, test suites and products.
- `iot-device-tester:DownloadTestSuite`

  Grants AWS IoT Device Tester permission to download test suite updates.

- `iot-device-tester:SendMetrics`

  Grants AWS permission to collect metrics about AWS IoT Device Tester internal usage.

(Optional) Install the AWS Command Line Interface

You might prefer to use the AWS CLI to perform some operations. If you don't have the AWS CLI installed, follow the instructions at Install the AWS CLI.

Configure the AWS CLI for the AWS Region you want to use by running **aws configure** from a command line. For information about the AWS Regions that support IDT for FreeRTOS, see AWS Regions and Endpoints. For more information about **aws configure** see Quick configuration with **aws configure**.

First test of your microcontroller board

You can use IDT for FreeRTOS to test as you port the FreeRTOS interfaces. After you have ported the FreeRTOS interfaces for your board's device drivers, you use AWS IoT Device Tester to run the qualification tests on your microcontroller board.

Add library porting layers

To port FreeRTOS for your device, follow the instructions in the FreeRTOS Porting Guide.

Configure your AWS credentials

You need to configure your AWS credentials for AWS IoT Device Tester to communicate with the AWS Cloud. For more information, see Set up AWS Credentials and Region for Development. Valid AWS credentials must be specified in the *devicetester_extract_location*/devicetester_afreertos_*[win|mac|linux]*/configs/config.json configuration file.

Topics

- Create a device pool in IDT for FreeRTOS
- Configure build, flash, and test settings

Create a device pool in IDT for FreeRTOS

Devices to be tested are organized in device pools. Each device pool consists of one or more identical devices. You can configure IDT for FreeRTOS to test a single device in a pool or multiple

devices in a pool. To accelerate the qualification process, IDT for FreeRTOS can test devices with the same specifications in parallel. It uses a round-robin method to execute a different test group on each device in a device pool.

You can add one or more devices to a device pool by editing the `devices` section of the `device.json` template in the `configs` folder.

> **Note**
>
> All devices in the same pool must be of same technical specification and SKU.

To enable parallel builds of the source code for different test groups, IDT for FreeRTOS copies the source code to a results folder inside the IDT for FreeRTOS extracted folder. The source code path in your build or flash command must be referenced using either the `testdata.sourcePath` or `sdkPath` variable. IDT for FreeRTOS replaces this variable with a temporary path of the copied source code. For more information see, IDT for FreeRTOS variables.

The following is an example `device.json` file used to create a device pool with multiple devices.

```
[
    {
        "id": "pool-id",
        "sku": "sku",
        "features": [
            {
                "name": "WIFI",
                "value": "Yes | No"
            },
            {
                "name": "Cellular",
                "value": "Yes | No"
            },
            {
                "name": "OTA",
                "value": "Yes | No",
                "configs": [
                    {
                        "name": "OTADataPlaneProtocol",
                        "value": "HTTP | MQTT"
                    }
                ]
```

```
            },
            {
                "name": "BLE",
                "value": "Yes | No"
            },
            {
                "name": "TCP/IP",
                "value": "On-chip | Offloaded | No"
            },
            {
                "name": "TLS",
                "value": "Yes | No"
            },
            {
                "name": "PKCS11",
                "value": "RSA | ECC | Both | No"
            },
            {
                "name": "KeyProvisioning",
                "value": "Import | Onboard | No"
            }
        ],

        "devices": [
          {
            "id": "device-id",
            "connectivity": {
              "protocol": "uart",
              "serialPort": "/dev/tty*"
            },
            **********Remove the section below if the device does not support onboard
key generation***************
            "secureElementConfig" : {
              "publicKeyAsciiHexFilePath": "absolute-path-to/public-key-txt-file:
contains-the-hex-bytes-public-key-extracted-from-onboard-private-key",
              "secureElementSerialNumber": "secure-element-serialNo-value",
              "preProvisioned"           : "Yes | No"
            },

**********************************************************************************************
            "identifiers": [
              {
                "name": "serialNo",
                "value": "serialNo-value"
```

```
                }
            ]
          }
        ]
    }
]
```

The following attributes are used in the `device.json` file:

`id`

A user-defined alphanumeric ID that uniquely identifies a pool of devices. Devices that belong to a pool must be of the same type. When a suite of tests is running, devices in the pool are used to parallelize the workload.

`sku`

An alphanumeric value that uniquely identifies the board you are testing. The SKU is used to track qualified boards.

> **Note**
>
> If you want to list your board in AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

`features`

An array that contains the device's supported features. AWS IoT Device Tester uses this information to select the qualification tests to run.

Supported values are:

`TCP/IP`

Indicates if your board supports a TCP/IP stack and whether it is supported on-chip (MCU) or offloaded to another module. TCP/IP is required for qualification.

`WIFI`

Indicates if your board has Wi-Fi capabilities. Must be set to `No` if `Cellular` is set to `Yes`.

`Cellular`

Indicates if your board has cellular capabilities. Must be set to `No` if `WIFI` is set to `Yes`. When this feature is set to `Yes`, the FullSecureSockets test will be executed using AWS t2.micro EC2 instances and this may incur additional costs to your account. For more information, see Amazon EC2 pricing.

`TLS`

Indicates if your board supports TLS. TLS is required for qualification.

`PKCS11`

Indicates the public key cryptography algorithm that the board supports. PKCS11 is required for qualification. Supported values are `ECC`, `RSA`, `Both` and `No`. `Both` indicates the board supports both the `ECC` and `RSA` algorithms.

`KeyProvisioning`

Indicates the method of writing a trusted X.509 client certificate onto your board. Valid values are `Import`, `Onboard` and `No`. Key provisioning is required for qualification.

- Use `Import` if your board allows the import of private keys. IDT will create a private key and build this to the FreeRTOS source code.
- Use `Onboard` if your board supports on-board private key generation (for example, if your device has a secure element, or if you prefer to generate your own device key pair and certificate). Make sure you add a `secureElementConfig` element in each of the device sections and put the absolute path to the public key file in the `publicKeyAsciiHexFilePath` field.
- Use `No` if your board does not support key provisioning.

`OTA`

Indicates if your board supports over-the-air (OTA) update functionality. The `OtaDataPlaneProtocol` attribute indicates which OTA dataplane protocol the device supports. The attribute is ignored if the OTA feature is not supported by the device. When `"Both"` is selected, the OTA test execution time is prolonged due to running both MQTT, HTTP, and mixed tests.

> **Note**
>
> Starting with IDT v4.1.0, `OtaDataPlaneProtocol` accepts only `HTTP` and `MQTT` as supported values.

`BLE`

Indicates if your board supports Bluetooth Low Energy (BLE).

`devices.id`

A user-defined unique identifier for the device being tested.

`devices.connectivity.protocol`

The communication protocol used to communicate with this device. Supported value: `uart`.

`devices.connectivity.serialPort`

The serial port of the host computer used to connect to the devices being tested.

`devices.secureElementConfig.PublicKeyAsciiHexFilePath`

The absolute path to the file that contains the hex bytes public key extracted from onboard private key.

Example format:

```
3059 3013 0607 2a86 48ce 3d02 0106 082a
8648 ce3d 0301 0703 4200 04cd 6569 ceb8
1bb9 1e72 339f e8cf 60ef 0f9f b473 33ac
6f19 1813 6999 3fa0 c293 5fae 08f1 1ad0
41b7 345c e746 1046 228e 5a5f d787 d571
dcb2 4e8d 75b3 2586 e2cc 0c
```

If your public key is in .der format, you can hex encode the public key directly to generate the hex file.

Example command for .der public key to generate hex file:

```
xxd -p pubkey.der > outFile
```

If your public key is in .pem format, you can extract the base64 encoded part, decode it into binary format, and then hex encode it to generate the hex file.

For example, use these commands to generate a hex file for a .pem public key:

1. Take out the base64 encoded part of the key (strip the header and footer) and store it in a file, for example name it `base64key`, run this command to convert it to .der format:

```
base64 –decode base64key > pubkey.der
```

2. Run the `xxd` command to convert it to hex format.

```
xxd -p pubkey.der > outFile
```

`devices.secureElementConfig.SecureElementSerialNumber`

(Optional) The serial number of the secure element. Provide this field when the serial number is printed out along with the device public key when you run the FreeRTOS demo/test project.

`devices.secureElementConfig.preProvisioned`

(Optional) Set to "Yes" if the device has a pre-provisioned secure element with locked-down credentials, that cannot import, create, or destroy objects. This configuration takes effect only when `features` has `KeyProvisioning` set to "Onboard", along with `PKCS11` set to "ECC".

`identifiers`

(Optional) An array of arbitrary name-value pairs. You can use these values in the build and flash commands described in the next section.

Configure build, flash, and test settings

For IDT for FreeRTOS to build and flash tests on to your board automatically, you must configure IDT to run the build and flash commands for your hardware. The build and flash command settings are configured in the `userdata.json` template file located in the `config` folder.

Configure settings for testing devices

Build, flash, and test settings are made in the `configs/userdata.json` file. We support Echo Server configuration by loading both the client and server certificates and keys in the

`customPath`. For more information, see Setting up an echo server in the *FreeRTOS Porting Guide*. The following JSON example shows how you can configure IDT for FreeRTOS to test multiple devices:

```
{
    "sourcePath": "/absolute-path-to/freertos",
    "vendorPath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-name",
    // ***********The sdkConfiguration block below is needed if you are not using the
 default, unmodified FreeRTOS repo.
    // In other words, if you are using the default, unmodified FreeRTOS repo then
 remove this block***************
    "sdkConfiguration": {
        "name": "sdk-name",
        "version": "sdk-version",
        "path": "/absolute-path-to/sdk"
    },
    "buildTool": {
        "name": "your-build-tool-name",
        "version": "your-build-tool-version",
        "command": [
            "{{config.idtRootPath}}/relative-path-to/build-parallel.sh
 {{testData.sourcePath}} {{enableTests}}"
        ]
    },
    "flashTool": {
        "name": "your-flash-tool-name",
        "version": "your-flash-tool-version",
        "command": [
            "/{{config.idtRootPath}}/relative-path-to/flash-parallel.sh
 {{testData.sourcePath}} {{device.connectivity.serialPort}} {{buildImageName}}"
        ],
        "buildImageInfo" : {
            "testsImageName": "tests-image-name",
            "demosImageName": "demos-image-name"
        }
    },
    "testStartDelayms": 0,
    "clientWifiConfig": {
        "wifiSSID": "ssid",
        "wifiPassword": "password",
        "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA |
 eWiFiSecurityWPA2 | eWiFiSecurityWPA3"
    },
```

```
    "testWifiConfig": {
        "wifiSSID": "ssid",
        "wifiPassword": "password",
        "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA |
 eWiFiSecurityWPA2 | eWiFiSecurityWPA3"
    },
    //**********
    //This section is used to start echo server based on server certificate generation
 method,
    //When certificateGenerationMethod is set as Automatic specify the eccCurveFormat
 to generate certifcate and key based on curve format,
    //When certificateGenerationMethod is set as Custom specify the certificatePath and
 PrivateKeyPath to be used to start echo server
    //**********
    "echoServerCertificateConfiguration": {
      "certificateGenerationMethod": "Automatic | Custom",
      "customPath": {
          "clientCertificatePath":"/path/to/clientCertificate",
          "clientPrivateKeyPath": "/path/to/clientPrivateKey",
          "serverCertificatePath":"/path/to/serverCertificate",
          "serverPrivateKeyPath": "/path/to/serverPrivateKey"
      },
    "eccCurveFormat": "P224 | P256 | P384 | P521"
    },
    "echoServerConfiguration": {
        "securePortForSecureSocket": 33333, // Secure tcp port used by SecureSocket
 test. Default value is 33333. Ensure that the port configured isn't blocked by the
 firewall or your corporate network
        "insecurePortForSecureSocket": 33334, // Insecure tcp port used by SecureSocket
 test. Default value is 33334. Ensure that the port configured isn't blocked by the
 firewall or your corporate network
        "insecurePortForWiFi": 33335 // Insecure tcp port used by Wi-Fi test. Default
 value is 33335. Ensure that the port configured isn't blocked by the firewall or your
 corporate network
    },
    "otaConfiguration": {
        "otaFirmwareFilePath": "{{testData.sourcePath}}/relative-path-to/ota-image-
generated-in-build-process",
        "deviceFirmwareFileName": "ota-image-name-on-device",
        "otaDemoConfigFilePath": "{{testData.sourcePath}}/relative-path-to/ota-demo-
config-header-file",
        "codeSigningConfiguration": {
            "signingMethod": "AWS | Custom",
            "signerHashingAlgorithm": "SHA1 | SHA256",
```

```
            "signerSigningAlgorithm": "RSA | ECDSA",
            "signerCertificate": "arn:partition:service:region:account-
id:resource:qualifier | /absolute-path-to/signer-certificate-file",
            "signerCertificateFileName": "signerCertificate-file-name",
            "compileSignerCertificate": boolean,
            // ***********Use signerPlatform if you choose aws for
 signingMethod***************
            "signerPlatform": "AmazonFreeRTOS-Default | AmazonFreeRTOS-TI-CC3220SF",
            "untrustedSignerCertificate": "arn:partition:service:region:account-
id:resourcetype:resource:qualifier",
            // ***********Use signCommand if you choose custom for
 signingMethod***************
            "signCommand": [
                "/absolute-path-to/sign.sh {{inputImageFilePath}}
 {{outputSignatureFilePath}}"
            ]
        }
    },
    // ***********Remove the section below if you're not configuring
 CMake***************
    "cmakeConfiguration": {
        "boardName": "board-name",
        "vendorName": "vendor-name",
        "compilerName": "compiler-name",
        "frToolchainPath": "/path/to/freertos/toolchain",
        "cmakeToolchainPath": "/path/to/cmake/toolchain"
    },
    "freertosFileConfiguration": {
        "required": [
            {
                "configName": "pkcs11Config",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_tests/config_files/core_pkcs11_config.h"
            },
            {
                "configName": "pkcs11TestConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_tests/config_files/iot_test_pkcs11_config.h"
            }
        ],
        "optional": [
            {
                "configName": "otaAgentTestsConfig",
```

```
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_tests/config_files/ota_config.h"
            },
            {
                "configName": "otaAgentDemosConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_demos/config_files/ota_config.h"
            },
            {
                "configName": "otaDemosConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_demos/config_files/ota_demo_config.h"
            }
        ]
    }
}
```

The following lists the attributes used in `userdata.json`:

`sourcePath`

The path to the root of the ported FreeRTOS source code. For parallel testing with an SDK, the `sourcePath` can be set using the `{{userData.sdkConfiguration.path}}` place holder. For example:

```
{ "sourcePath":"{{userData.sdkConfiguration.path}}/freertos" }
```

`vendorPath`

The path to the vendor specific FreeRTOS code. For serial testing, the `vendorPath` can be set as an absolute path. For example:

```
{ "vendorPath":"C:/path-to-freertos/vendors/espressif/boards/esp32" }
```

For parallel testing, the `vendorPath` can be set using the `{{testData.sourcePath}}` place holder. For example:

```
{ "vendorPath":"{{testData.sourcePath}}/vendors/espressif/boards/esp32" }
```

The `vendorPath` variable is only necessary when running without an SDK, it can be removed otherwise.

> **Note**
>
> When running tests in parallel without an SDK, the `{{testData.sourcePath}}` placeholder must be used in the `vendorPath`, `buildTool`, `flashTool` fields. When running test with a single device, absolute paths must be used in the `vendorPath`, `buildTool`, `flashTool` fields. When running with an SDK, the `{{sdkPath}}` placeholder must be used in the `sourcePath`, `buildTool`, and `flashTool` commands.

`sdkConfiguration`

If you are qualifying FreeRTOS with any modifications to files and folder structure beyond what is required for porting, then you will need to configure your SDK information in this block. If you're not qualifying with a ported FreeRTOS inside of an SDK, then you should omit this block entirely.

`sdkConfiguration.name`

The name of the SDK you're using with FreeRTOS. If you're not using an SDK, then the entire `sdkConfiguration` block should be omitted.

`sdkConfiguration.version`

The version of the SDK you're using with FreeRTOS. If you're not using an SDK, then the entire `sdkConfiguration` block should be omitted.

`sdkConfiguration.path`

The absolute path to your SDK directory that contains your FreeRTOS code. If you're not using an SDK, then the entire `sdkConfiguration` block should be omitted.

`buildTool`

The full path to your build script (.bat or .sh) that contains the commands to build your source code. All references to the source code path in the build command must be replaced by the AWS IoT Device Tester variable `{{testdata.sourcePath}}` and references to the SDK path should be replaced by `{{sdkPath}}`. Use the `{{config.idtRootPath}}` placeholder to reference the absolute or relative IDT path.

`testStartDelayms`

Specifies how many milliseconds the FreeRTOS test runner will wait before starting to run tests. This can be useful if the device under test begins outputting important test information before IDT has a chance to connect and start logging due to network or other latency. The max allowed value is 30000 ms (30 seconds). This value is applicable to FreeRTOS test groups only, and not applicable to other test groups that do not utilize the FreeRTOS test runner, such as the OTA tests.

`flashTool`

Full path to your flash script (.sh or .bat) that contains the flash commands for your device. All references to the source code path in the flash command must be replaced by the IDT for FreeRTOS variable `{{testdata.sourcePath}}` and all references to your SDK path must be replaced by the IDT for FreeRTOS variable `{{sdkPath}}`.Use the `{{config.idtRootPath}}` placeholder to reference the absolute or relative IDT path.

`buildImageInfo`

`testsImageName`

The name of the file produced by the build command when building tests from the *`freertos-source`*`/tests` folder.

`demosImageName`

The name of the file produced by the build command when building tests from the *`freertos-source`*`/demos` folder.

`clientWifiConfig`

The client Wi-Fi configuration. The Wi-Fi library tests require an MCU board to connect to two access points. (The two access points can be the same.) This attribute configures the Wi-Fi settings for the first access point. Some of the Wi-Fi test cases expect the access point to have some security and not to be open. Please make sure both access points are on the same subnet as the host computer running IDT.

`wifi_ssid`

The Wi-Fi SSID.

`wifi_password`

The Wi-Fi password.

`wifiSecurityType`

The type of Wi-Fi security used. One of the values:

- `eWiFiSecurityOpen`
- `eWiFiSecurityWEP`
- `eWiFiSecurityWPA`
- `eWiFiSecurityWPA2`
- `eWiFiSecurityWPA3`

> **Note**
>
> If your board does not support Wi-Fi, you must still include the `clientWifiConfig` section in your `device.json` file, but you can omit values for these attributes.

`testWifiConfig`

The test Wi-Fi configuration. The Wi-Fi library tests require an MCU board to connect to two access points. (The two access points can be the same.) This attribute configures the Wi-Fi setting for the second access point. Some of the Wi-Fi test cases expect the access point to have some security and not to be open. Please make sure both access points are on the same subnet as the host computer running IDT.

`wifiSSID`

The Wi-Fi SSID.

`wifiPassword`

The Wi-Fi password.

`wifiSecurityType`

The type of Wi-Fi security used. One of the values:

- `eWiFiSecurityOpen`
- `eWiFiSecurityWEP`
- `eWiFiSecurityWPA`
- `eWiFiSecurityWPA2`

- `eWiFiSecurityWPA3`

> **Note**
>
> If your board does not support Wi-Fi, you must still include the `testWifiConfig` section in your `device.json` file, but you can omit values for these attributes.

`echoServerCertificateConfiguration`

The configurable echo server certificate generation placeholder for secure socket tests. This field is required.

`certificateGenerationMethod`

Specifies whether the server certificate is generated automatically or provided manually.

`customPath`

If `certificateGenerationMethod` is "Custom", `certificatePath` and `privateKeyPath` are required.

`certificatePath`

Specifies the filepath for the server certificate.

`privateKeyPath`

Specifies the filepath for the private key.

`eccCurveFormat`

Specifies the curve format supported by the board. Required when `PKCS11` is set to "ecc" in `device.json`. Valid values are "P224", "P256", "P384", or "P521".

`echoServerConfiguration`

The configurable echo server ports for WiFi and secure sockets tests. This field is optional.

`securePortForSecureSocket`

The port which is used to setup an echo server with TLS for the secure sockets test. The default value is 33333. Ensure the port configured is not blocked by a firewall or your corporate network.

`insecurePortForSecureSocket`

The port which is used to setup echo server without TLS for the secure sockets test. The default value used in the test is 33334. Ensure the port configured is not blocked by a firewall or your corporate network.

`insecurePortForWiFi`

The port which is used to setup echo server without TLS for WiFi test. The default value used in the test is 33335. Ensure the port configured is not blocked by a firewall or your corporate network.

`otaConfiguration`

The OTA configuration. [Optional]

`otaFirmwareFilePath`

The full path to the OTA image created after the build. For example, `{{testData.sourcePath}}/`*`relative-path/to/ota/image/from/source/root`*.

`deviceFirmwareFileName`

The full file path on the MCU device where the OTA firmware is located. Some devices do not use this field, but you still must provide a value.

`otaDemoConfigFilePath`

The full path to `aws_demo_config.h`, found in *`afr-source`*`/vendors/vendor/boards/board/aws_demos/config_files/`. These files are included in the porting code template that FreeRTOS provides.

`codeSigningConfiguration`

The code signing configuration.

`signingMethod`

The code signing method. Possible values are `AWS` or `Custom`.

> **Note**
>
> For the Beijing and Ningxia Regions, use `Custom`. AWS code signing isn't supported in these Regions.

`signerHashingAlgorithm`

The hashing algorithm supported on the device. Possible values are `SHA1` or `SHA256`.

`signerSigningAlgorithm`

The signing algorithm supported on the device. Possible values are `RSA` or `ECDSA`.

`signerCertificate`

The trusted certificate used for OTA.

For AWS code signing method, use the Amazon Resource Name (ARN) for the trusted certificate uploaded to the AWS Certificate Manager.

For Custom code signing method, use the absolute path to the signer certificate file.

For more information about creating a trusted certificate, see Create a code-signing certificate.

`signerCertificateFileName`

The file name of the code signing certificate on the device. This value must match the file name that you provided when you ran the `aws acm import-certificate` command.

For more information, see Create a code-signing certificate.

`compileSignerCertificate`

Set to `true` if the code signer signature verification certificate isn't provisioned or flashed, so it must be compiled into the project. AWS IoT Device Tester fetches the trusted certificate and compiles it into `aws_codesigner_certifiate.h`.

`untrustedSignerCertificate`

The ARN or filepath for a second certificate used in some OTA tests as an untrusted certificate. For more information about creating a certificate, see Create a code-signing certificate.

`signerPlatform`

The signing and hashing algorithm that AWS Code Signer uses while creating the OTA update job. Currently, the possible values for this field are `AmazonFreeRTOS-TI-CC3220SF` and `AmazonFreeRTOS-Default`.

- Choose `AmazonFreeRTOS-TI-CC3220SF` if `SHA1` and `RSA`.
- Choose `AmazonFreeRTOS-Default` if `SHA256` and `ECDSA`.

If you need `SHA256` | `RSA` or `SHA1` | `ECDSA` for your configuration, contact us for further support.

Configure `signCommand` if you chose `Custom` for `signingMethod`.

`signCommand`

The command used to perform custom code signing. You can find the template in the `/configs/script_templates` directory.

Two placeholders `{{inputImageFilePath}}` and `{{outputSignatureFilePath}}` are required in the command. `{{inputImageFilePath}}` is the file path of the image built by IDT to be signed. `{{outputSignatureFilePath}}` is the file path of the signature which will be generated by the script.

`cmakeConfiguration`

CMake configuration [Optional]

> **Note**
>
> To execute CMake test cases, you must provide the board name, vendor name, and either the `frToolchainPath` or `compilerName`. You may also provide the `cmakeToolchainPath` if you have a custom path to the CMake toolchain.

`boardName`

The name of the board under test. The board name should be the same as the folder name under *`path/to/afr/source/code`*`/vendors/`*`vendor`*`/boards/`*`board`*.

`vendorName`

The vendor name for the board under test. The vendor should be the same as the folder name under *`path/to/afr/source/code`*`/vendors/`*`vendor`*.

`compilerName`

The compiler name.

`frToolchainPath`

The fully-qualified path to the compiler toolchain

`cmakeToolchainPath`

The fully-qualified path to the CMake toolchain. This field is optional

`freertosFileConfiguration`

The configuration of the FreeRTOS files that IDT modifies before running tests.

`required`

This section specifies required tests whose config files you have moved, for example, PKCS11, TLS, and so on.

`configName`

The name of the test that is being configured.

`filePath`

The absolute path to the configuration files within the *`freertos`* repo. Use the `{{testData.sourcePath}}` variable to define the path.

`optional`

This section specifies optional tests whose config files you have moved, for example OTA, WiFi, and so on.

`configName`

The name of the test that is being configured.

`filePath`

The absolute path to the configuration files within the *`freertos`* repo. Use the `{{testData.sourcePath}}` variable to define the path.

> **Note**
>
> To execute CMake test cases, you must provide the board name, vendor name, and either the `afrToolchainPath` or `compilerName`. You may also provide `cmakeToolchainPath` if you have a custom path to the CMake toolchain.

IDT for FreeRTOS variables

The commands to build your code and flash the device might require connectivity or other information about your devices to run successfully. AWS IoT Device Tester allows you to reference device information in flash and build commands using JsonPath. By using simple JsonPath expressions, you can fetch the required information specified in your `device.json` file.

Path variables

IDT for FreeRTOS defines the following path variables that can be used in command lines and configuration files:

`{{testData.sourcePath}}`

Expands to the source code path. If you use this variable, it must be used in both the flash and build commands.

`{{sdkPath}}`

Expands to the value in your `userData.sdkConfiguration.path` when used in the build and flash commands.

`{{device.connectivity.serialPort}}`

Expands to the serial port.

`{{device.identifiers[?(@.name == 'serialNo')].value[0]}}`

Expands to the serial number of your device.

`{{enableTests}}`

Integer value indicating whether the build is for tests (value 1) or demos (value 0).

`{{buildImageName}}`

The file name of the image built by the build command.

`{{otaCodeSignerPemFile}}`

PEM file for the OTA code signer.

`{{config.idtRootPath}}`

Expands to the AWS IoT Device Tester root path. This variable replaces the absolute path for IDT when used by the build and flash commands.

Use the IDT user interface to run the FreeRTOS qualification suite

Starting with IDT v4.3.0, AWS IoT Device Tester for FreeRTOS (IDT-FreeRTOS) includes a web-based user interface that enables you to interact with the IDT command line executable and related configuration files. You can use the IDT-FreeRTOS UI to create a new configuration to run IDT tests, or to modify an existing configuration. You can also use the UI to invoke the IDT executable and run tests.

The IDT-FreeRTOS UI provides the following functions:

- Simplify setting up configuration files for IDT-FreeRTOS tests.
- Simplify using IDT-FreeRTOS to run qualification tests.

For information about the using the command line to run qualification tests, see First test of your microcontroller board.

This section describes the prerequisites for using the IDT-FreeRTOS UI, and shows you how to get started running qualification tests in the UI.

Topics

- Set up the prerequisites to run the FreeRTOS qualification suite
- Get started with the IDT-FreeRTOS UI

Set up the prerequisites to run the FreeRTOS qualification suite

This section describes the prerequisites for testing microcontrollers with AWS IoT Device Tester.

Topics

- Use a supported web browser
- Download FreeRTOS
- Download IDT for FreeRTOS
- Create and configure an AWS account
- AWS IoT Device Tester managed policy

Use a supported web browser

The IDT-FreeRTOS UI supports the following web browsers.

Browser	Version
Google Chrome	Latest three major versions
Mozilla Firefox	Latest three major versions
Microsoft Edge	Latest three major versions
Apple Safari for macOS	Latest three major versions

We recommend that you use Google Chrome or Mozilla Firefox for a better experience.

> **Note**
>
> The IDT-FreeRTOS UI doesn't support Microsoft Internet Explorer.

Download FreeRTOS

You can download a release of FreeRTOS from GitHub with the following command:

```
git clone --branch <FREERTOS_RELEASE_VERSION> --recurse-submodules https://github.com/
aws/amazon-freertos.git
cd amazon-freertos
git submodule update --checkout --init --recursive
```

where <FREERTOS_RELEASE_VERSION> is a version of FreeRTOS (for example, 202007.00) corresponding to an IDT version listed in Supported versions of AWS IoT Device Tester. This ensures you have the full source code, including submodules, and are using the correct version of IDT for your version of FreeRTOS, and vice versa.

Windows has a path length limitation of 260 characters. The path structure of FreeRTOS is many levels deep, so if you're using Windows, keep your file paths under the 260-character limit. For example, clone FreeRTOS to `C:\FreeRTOS` rather than `C:\Users\username\programs\projects\myproj\FreeRTOS\`.

Considerations for LTS qualification (qualification for FreeRTOS that uses LTS libraries)

- In order for your microcontroller to be designated as supporting long-term support (LTS) based versions of FreeRTOS in the AWS Partner Device Catalog, you must provide a manifest file. For more information, see the FreeRTOS Qualification Checklist in the *FreeRTOS Qualification Guide*.
- In order to validate that your microcontroller supports LTS based versions of FreeRTOS and qualify it for submission to the AWS Partner Device Catalog, you must use AWS IoT Device Tester (IDT) with FreeRTOS Qualification (FRQ) test suite version v1.4.x.
- Support for LTS based versions of FreeRTOS is limited to the 202012.xx version of FreeRTOS.

Download IDT for FreeRTOS

Every version of FreeRTOS has a corresponding version of IDT for FreeRTOS for performing qualification tests. Download the appropriate version of IDT for FreeRTOS from Supported versions of AWS IoT Device Tester.

Extract IDT for FreeRTOS to a location on the file system where you have read and write permissions. Because Microsoft Windows has a character limit for the path length, extract IDT for FreeRTOS into a root directory such as `C:\` or `D:\`.

> **Note**
>
> We recommend that you extract the IDT package to a local drive.Allowing multiple users to run IDT from a shared location, such as an NFS directory or a Windows network shared folder, might result in the system not responding or data corruption.

Create and configure an AWS account

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

AWS IoT Device Tester managed policy

To enable device tester to run and to collect metrics, the `AWSIoTDeviceTesterForFreeRTOSFullAccess` managed policy contains the following permissions:

- `iot-device-tester:SupportedVersion`

  Grants permission to get the list of FreeRTOS versions and test suite versions supported by IDT, so that they're available from the AWS CLI.
- `iot-device-tester:LatestIdt`

  Grants permission to get the latest AWS IoT Device Tester version that is available for download.
- `iot-device-tester:CheckVersion`

Grants permission to check that a combination of product, test suite, and AWS IoT Device Tester versions are compatible.

- `iot-device-tester:DownloadTestSuite`

  Grants permission to AWS IoT Device Tester to download test suites.

- `iot-device-tester:SendMetrics`

  Grants permission to publish AWS IoT Device Tester usage metrics data.

Get started with the IDT-FreeRTOS UI

This section shows you how to use the IDT-FreeRTOS UI to create or modify your configuration, and then shows you how to run tests.

Topics

- Configure AWS credentials
- Open the IDT-FreeRTOS UI
- Create a new configuration
- Modify an existing configuration
- Run qualification tests

Configure AWS credentials

You must configure credentials for the AWS user that you created in Create and configure an AWS account. You can specify your credentials in one of two ways:

- In a credentials file
- As environment variables

Configure AWS credentials with a credentials file

IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

The location of the credentials file varies, depending on the operating system you're using:

- macOS, Linux: `~/.aws/credentials`
- Windows: `C:\Users\UserName\.aws\credentials`

Add your AWS credentials to the `credentials` file in the following format:

```
[default]
aws_access_key_id = <your_access_key_id>
aws_secret_access_key = <your_secret_access_key>
```

> **Note**
>
> If you don't use the `default` AWS profile, be sure to specify the profile name in the IDT-FreeRTOS UI. For more information about profiles, see Configuration and credential file settings.

Configure AWS credentials with environment variables

Environment variables are variables maintained by the operating system and used by system commands. They're not saved if you close the SSH session. The IDT-FreeRTOS UI uses the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to store your AWS credentials.

To set these variables on Linux, macOS, or Unix, use **export**:

```
export AWS_ACCESS_KEY_ID=<your_access_key_id>
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

To set these variables on Windows, use **set**:

```
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

Open the IDT-FreeRTOS UI

To open the IDT-FreeRTOS UI

1. Download a supported IDT-FreeRTOS version and extract the downloaded archive into a location on your file system where you have read and write permissions.

2. Run the following command to navigate to the IDT-FreeRTOS installation directory:

```
cd devicetester-extract-location/bin
```

3. Run the following command to open the IDT-FreeRTOS UI:

   Linux

   ```
   .devicetestergui_linux_x86-64.exe
   ```

   Windows

   ```
   ./devicetestergui_win_x64-64
   ```

   macOS

   ```
   ./devicetestergui_mac_x86-64
   ```

   > **Note**
   >
   > On Mac, to allow your system to run the UI, go to **System Preferences -> Security & Privacy**. When you run the tests, you may need to do this three more times.

   The IDT-FreeRTOS UI opens in your default browser. For information about supported browsers, see Use a supported web browser.

Create a new configuration

If you're a first-time user, then you must create a new configuration to set up the JSON configuration files that IDT-FreeRTOS requires to run tests. You can then run tests or modify the configuration that was created.

For examples of the `config.json`, `device.json`, and `userdata.json` files, see First test of your microcontroller board. For an example of the `resource.json` file that is used only for running Bluetooth Low Energy (BLE) tests, see Run Bluetooth Low Energy tests.

To create a new configuration

1. In the IDT-FreeRTOS UI, open the navigation menu, and then choose **Create new configuration**.

   > **Important**
   >
   > You must configure your AWS credentials before you open the UI. If you haven't configured your credentials, close the IDT-FreeRTOS UI browser window, follow the steps in Configure AWS credentials, and then reopen the IDT-FreeRTOS UI.

2. Follow the configuration wizard to enter the IDT configuration settings that are used to run qualification tests. The wizard configures the following settings in JSON configuration files that are located in the *devicetester-extract-location*`/config` directory.

   - **AWS settings**—The AWS account information that IDT-FreeRTOS uses to create AWS resources during test runs. These settings are configured in the `config.json` file.
   - **FreeRTOS repository**—The absolute path to the FreeRTOS repository and ported code, and the type of qualification you want to perform. These settings are configured in the `userdata.json` file.

     You must port FreeRTOS for your device before you can run qualification tests. For more information, see the FreeRTOS Porting Guide
   - **Build and flash**—The build and flash commands for your hardware that allow IDT to build and flash tests on to your board automatically. These settings are configured in the `userdata.json` file.
   - **Devices**—The device pool settings for the devices to be tested. These settings are configured in `id` and `sku` fields, and the `devices` block for the device pool in the `device.json` file.
   - **Networking**—The settings to test network communication support for your devices. These settings are configured in the `features` block of the `device.json` file, and in the `clientWifiConfig` and `testWifiConfig` blocks in the `userdata.json` file.
   - **Echo server**—The echo server configuration settings for secure socket tests. These settings are configured in the `userdata.json` file.

     For more information about echo server configuration, see https://docs.aws.amazon.com/freertos/latest/portingguide/afr-echo-server.html.
   - **CMake**—(Optional) The settings to run CMake build functionality tests. This configuration is required only if you're using CMake as your build system. These settings are configured in the `userdata.json` file.

- **BLE**—The settings to run Bluetooth Low Energy functionality tests. These settings are configured in the `features` block of the `device.json` file and in the `resource.json` file.
- **OTA**—The settings to run OTA functionality tests. These settings are configured in the `features` block of the `device.json` file and in the `userdata.json` file.

3. On the **Review** page, verify your configuration information.

After you finish reviewing your configuration, to run your qualification tests, choose **Run tests**.

Modify an existing configuration

If you have already set up configuration files for IDT, then you can use the IDT-FreeRTOS UI to modify your existing configuration. Make sure that your existing configuration files are available in the *devicetester-extract-location*`/config` directory.

To modify a new configuration

1. In the IDT-FreeRTOS UI, open the navigation menu, and then choose **Edit existing configuration**.

   The configuration dashboard displays information about your existing configuration settings. If a configuration is incorrect or unavailable, the status for that configuration is `Error validating configuration`.
2. To modify an existing configuration setting, complete the following steps:

   a. Choose the name of a configuration setting to open its settings page.
   b. Modify the settings, and then choose **Save** to regenerate the corresponding configuration file.

After you finish modifying your configuration, verify that all of your configuration settings pass validation. If the status for each configuration setting is `Valid`, you can run your qualification tests using this configuration.

Run qualification tests

After you have created a configuration for IDT-FreeRTOS, you can run your qualification tests.

To run qualification tests

1. Validate your configuration.

2. In the navigation menu, choose **Run tests**.
3. To start the test run, choose **Start tests**.

IDT-FreeRTOS runs the qualification tests, and displays the test run summary and any errors in the **Test runner** console. After the test run is complete, you can view the test results and logs from the following locations:

- Test results are located in the *devicetester-extract-location*/results/*execution-id* directory.
- Test logs are located in the *devicetester-extract-location*/results/*execution-id*/logs directory.

For more information about test results and logs, see View the IDT for FreeRTOS results and View the IDT for FreeRTOS logs.

Run Bluetooth Low Energy tests

This section describes how to set up and run Bluetooth Low Energy tests using AWS IoT Device Tester for FreeRTOS.

Bluetooth tests are not required for core qualification. If you do not want to test your device with FreeRTOS Bluetooth support you may skip this setup, be sure to leave the BLE feature in device.json set to No.

Prerequisites

- Follow the instructions in First test of your microcontroller board.
- A Raspberry Pi 4B or 3B+. (Required to run the Raspberry Pi BLE companion application)
- A micro SD card and SD card adapter for the Raspberry Pi software.

Raspberry Pi setup

To test the BLE capabilities of the device under test (DUT), you must have a Raspberry Pi Model 4B or 3B+.

To set up your Raspberry Pi to run BLE tests

1. Download one of the custom Yocto images that contains the software required to perform the tests.

   - Image for Raspberry Pi 4B
   - Image for Raspberry Pi 3B+

   > **Note**
   >
   > The Yocto image should only be used for testing with AWS IoT Device Tester for FreeRTOS and not for any other purpose.

2. Flash the yocto image onto the SD card for Raspberry Pi.

   - Using an SD card-writing tool such as Etcher, flash the downloaded *image-name*`.rpi-sd.img` file onto the SD card. Because the operating system image is large, this step might take some time. Then eject your SD card from your computer and insert the microSD card into your Raspberry Pi.

3. Configure your Raspberry Pi.

   a. For the first boot, we recommend that you connect the Raspberry Pi to a monitor, keyboard, and mouse.

   b. Connect your Raspberry Pi to a micro USB power source.

   c. Sign in using the default credentials. For user ID, enter **`root`**. For password, enter **`idtafr`**.

   d. Using an Ethernet or Wi-Fi connection, connect the Raspberry Pi to your network.

      i. To connect your Raspberry Pi over Wi-Fi, open `/etc/wpa_supplicant.conf` on the Raspberry Pi and add your Wi-Fi credentials to the `Network` configuration.

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
update_config=1

network={
        scan_ssid=1
        ssid="your-wifi-ssid"
        psk="your-wifi-password"
```

```
        }
```

ii. Run `ifup wlan0` to start the Wi-Fi connection. It might take a minute to connect to your Wi-Fi network.

e. For an Ethernet connection, run `ifconfig eth0`. For a Wi-Fi connection, run `ifconfig wlan0`. Make a note of the IP address, which appears as `inet addr` in the command output. You need the IP address later in this procedure.

f. (Optional) The tests execute commands on the Raspberry Pi over SSH using the default credentials for the yocto image. For additional security, we recommend that you set up public key authentication for SSH and disable password-based SSH.

i. Create an SSH key using the OpenSSL `ssh-keygen` command. If you already have an SSK key pair on your host computer, it is a best practice to create a new one to allow AWS IoT Device Tester for FreeRTOS to sign in to your Raspberry Pi.

> **Note**
>
> Windows does not come with an installed SSH client. For information about how to install an SSH client on Windows, see Download SSH Software.

ii. The `ssh-keygen` command prompts you for a name and path to store the key pair. By default, the key pair files are named `id_rsa` (private key) and `id_rsa.pub` (public key). On macOS and Linux, the default location of these files is `~/.ssh/`. On Windows, the default location is `C:\Users\`*`user-name`*.

iii. When you are prompted for a key phrase, just press ENTER to continue.

iv. To add your SSH key onto your Raspberry Pi so AWS IoT Device Tester for FreeRTOS can sign into the device, use the `ssh-copy-id` command from your host computer. This command adds your public key into the `~/.ssh/authorized_keys` file on your Raspberry Pi.

```
ssh-copy-id root@raspberry-pi-ip-address
```

v. When prompted for a password, enter **`idtafr`**. This is the default password for the yocto image.

> **Note**
>
> The `ssh-copy-id` command assumes the public key is named `id_rsa.pub`. On macOS and Linux, the default location is `~/.ssh/`. On Windows, the default location is `C:\Users\`*`user-name`*`\.ssh`. If you gave the public key a different name or stored it in a different location, you must specify the fully qualified path to your SSH public key using the `-i` option to `ssh-copy-id` (for example, `ssh-copy-id -i ~/my/path/myKey.pub`). For more information about creating SSH keys and copying public keys, see SSH-COPY-ID.

vi. To test that the public key authentication is working, run `ssh -i` */my/path/myKey* `root@`*`raspberry-pi-device-ip`*.

If you are not prompted for a password, your public key authentication is working.

vii. Verify that you can sign in to your Raspberry Pi using a public key, and then disable password-based SSH.

A. On the Raspberry Pi, edit the `/etc/ssh/sshd_config` file.

B. Set the `PasswordAuthentication` attribute to no.

C. Save and close the `sshd_config` file.

D. Reload the SSH server by running `/etc/init.d/sshd reload`.

g. Create a `resource.json` file.

i. In the directory in which you extracted AWS IoT Device Tester, create a file named `resource.json`.

ii. Add the following information about your Raspberry Pi to the file, replacing *`raspberry-pi-ip-address`* with the IP address of your Raspberry Pi.

```
[
    {
        "id": "ble-test-raspberry-pi",
        "features": [
            {"name":"ble", "version":"4.2"}
        ],
        "devices": [
            {
```

```
                "id": "ble-test-raspberry-pi-1",
                "connectivity": {
                    "protocol": "ssh",
                    "ip": "rasp-pi-ip-address"
                }
            }
        ]
    }
]
```

iii. If you didn't choose to use public key authentication for SSH, add the following to the `connectivity` section of the `resource.json` file.

```
"connectivity": {
    "protocol": "ssh",
    "ip": "rasp-pi-ip-address",
    "auth": {
        "method": "password",
        "credentials": {
            "user": "root",
            "password": "idtafr"
        }
    }
}
```

iv. (Optional) If you chose to use public key authentication for SSH, add the following to the `connectivity` section of the `resource.json` file.

```
"connectivity": {
    "protocol": "ssh",
    "ip": "rasp-pi-ip-address",
    "auth": {
        "method": "pki",
        "credentials": {
            "user": "root",
            "privKeyPath": "location-of-private-key"
        }
    }
}
```

FreeRTOS device setup

In your `device.json` file, set the BLE feature to `Yes`. If you are starting with a `device.json` file from before Bluetooth tests were available, you need to add the feature for BLE to the `features` array:

```
{
    ...
    "features": [
        {
            "name": "BLE",
            "value": "Yes"
        },
    ...
}
```

Run the BLE tests

After you have enabled the BLE feature in `device.json`, the BLE tests run when you run `devicetester_`*`[linux | mac | win_x86-64]`* `run-suite` *without specifying a group-id*.

If you want to run the BLE tests separately, you can specify the group ID for BLE: `devicetester_`*`[linux | mac | win_x86-64]`* `run-suite --userdata` *`path-to-userdata`*`/userdata.json --group-id FullBLE`.

For the most reliable performance, place your Raspberry Pi close to the device under test (DUT).

Troubleshoot BLE tests

Make sure you have followed the steps in First test of your microcontroller board. If tests other than BLE are failing, then the problem is most likely not due to the Bluetooth configuration.

Run the FreeRTOS qualification suite

You use the AWS IoT Device Tester for FreeRTOS executable to interact with IDT for FreeRTOS. The following command line examples show you how to run the qualification tests for a device pool (a set of identical devices).

IDT v3.0.0 and later

```
devicetester_[linux | mac | win] run-suite  \
```

```
    --suite-id suite-id  \
    --group-id group-id  \
    --pool-id your-device-pool \
    --test-id test-id  \
    --upgrade-test-suite y|n  \
    --update-idt y|n  \
    --update-managed-policy y|n  \
    --userdata userdata.json
```

Runs a suite of tests on a pool of devices. The `userdata.json` file must be located in the *devicetester_extract_location*/devicetester_afreertos_*[win|mac|linux]*/`configs/` directory.

> **Note**
>
> If you're running IDT for FreeRTOS on Windows, use forward slashes (/) to specify the path to the `userdata.json` file.

Use the following command to run a specific test group:

```
devicetester_[linux | mac | win] run-suite  \
    --suite-id FRQ_1.0.1  \
    --group-id group-id  \
    --pool-id pool-id  \
    --userdata userdata.json
```

The `suite-id` and `pool-id` parameters are optional if you're running a single test suite on a single device pool (that is, you have only one device pool defined in your `device.json` file).

Use the following command to run a specific test case in a test group:

```
devicetester_[linux | mac | win_x86-64] run-suite  \
    --group-id group-id  \
    --test-id test-id
```

You can use the `list-test-cases` command to list the test cases in a test group.

IDT for FreeRTOS command line options

group-id

(Optional) The test groups to run, as a comma-separated list. If not specified, IDT runs all test groups in the test suite.

pool-id

(Optional) The device pool to test. This is required if you define multiple device pools in `device.json`. If you only have one device pool, you can omit this option.

suite-id

(Optional) The test suite version to run. If not specified, IDT uses the latest version in the tests directory on your system.

> **Note**
>
> Starting in IDT v3.0.0, IDT checks online for newer test suites. For more information, see Test suite versions.

test-id

(Optional) The tests to run, as a comma-separated list. If specified, `group-id` must specify a single group.

Example

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id mqtt --test-id
 mqtt_test
```

update-idt

(Optional) If this parameter is not set and a newer IDT version is available, you will be prompted to update IDT. If this parameter is set to `Y`, IDT will stop test execution if it detects that a newer version is available. If this parameter is set to `N`, IDT will continue the test execution.

update-managed-policy

(Optional) If this parameter is not used and IDT detects that your managed policy isn't up-to-date, you will be prompted to update your managed policy. If this parameter is set to `Y`,

IDT will stop test execution if it detects that your managed policy isn't up-to-date. If this parameter is set to `N`, IDT will continue the test execution.

upgrade-test-suite

(Optional) If not used, and a newer test suite version is available, you're prompted to download it. To hide the prompt, specify `y` to always download the latest test suite, or `n` to use the test suite specified or the latest version on your system.

Example

Example

To always download and use the latest test suite, use the following command.

```
devicetester_[linux | mac | win_x86-64] run-suite --userdata userdata file --
group-id group ID --upgrade-test-suite y
```

To use the latest test suite on your system, use the following command.

```
devicetester_[linux | mac | win_x86-64] run-suite --userdata userdata file --
group-id group ID --upgrade-test-suite n
```

h

Use the help option to learn more about `run-suite` options.

Example

Example

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

IDT v1.7.0 and earlier

```
devicetester_[linux | mac | win] run-suite  \
    --suite-id suite-id  \
    --pool-id your-device-pool  \
    --userdata userdata.json
```

The `userdata.json` file should be located in the *devicetester_extract_location*/`devicetester_afreertos_`*[win|mac|linux]*`/configs/` directory.

> **Note**
>
> If you are running IDT for FreeRTOS on Windows, use forward slashes (/) to specify the path to the `userdata.json` file.

Use the following command to run a specific test group.

```
devicetester_[linux | mac | win] run-suite  \
    --suite-id FRQ_1 --group-id group-id  \
    --pool-id pool-id  \
    --userdata userdata.json
```

`suite-id` and `pool-id` are optional if you are running a single test suite on a single device pool (that is, you have only one device pool defined in your `device.json` file).

IDT for FreeRTOS command line options

group-id

(Optional) Specifies the test group.

pool-id

Specifies the device pool to test. If you only have one device pool, you can omit this option.

suite-id

(Optional) Specifies the test suite to run.

IDT for FreeRTOS commands

The IDT for FreeRTOS command supports the following operations:

IDT v3.0.0 and later

`help`

Lists information about the specified command.

`list-groups`

Lists the groups in a given suite.

`list-suites`

Lists the available suites.

`list-supported-products`

Lists the supported products and test suite versions.

`list-supported-versions`

Lists the FreeRTOS and test suite versions supported by the current IDT version.

`list-test-cases`

Lists the test cases in a specified group.

`run-suite`

Runs a suite of tests on a pool of devices.

Use the `--suite-id` option to specify a test suite version, or omit it to use the latest version on your system.

Use the `--test-id` to run an individual test case.

Example

```
devicetester_[linux | mac | win_x86-64] run-suite --group-id mqtt --test-id
 mqtt_test
```

For a complete list of options see Run the FreeRTOS qualification suite.

> **Note**
>
> Starting in IDT v3.0.0, IDT checks online for newer test suites. For more information, see Test suite versions.

IDT v1.7.0 and earlier

`help`

Lists information about the specified command.

`list-groups`

Lists the groups in a given suite.

`list-suites`

Lists the available suites.

`run-suite`

Runs a suite of tests on a pool of devices.

Test for re-qualification

As new versions of IDT for FreeRTOS qualification tests are released, or as you update your board-specific packages or device drivers, you can use IDT for FreeRTOS to test your microcontroller boards. For subsequent qualifications, make sure that you have the latest versions of FreeRTOS and IDT for FreeRTOS and run the qualification tests again.

View the IDT for FreeRTOS results

While running, IDT writes errors to the console, log files, and test reports. After IDT completes the qualification test suite, it writes a test run summary to the console and generates two test reports. These reports can be found in *`devicetester-extract-location`*`/results/`*`execution-id`*`/`. Both reports capture the results from the qualification test suite execution.

The `awsiotdevicetester_report.xml` is the qualification test report that you submit to AWS to list your device in the AWS Partner Device Catalog. The report contains the following elements:

- The IDT for FreeRTOS version.
- The FreeRTOS version that was tested.
- The features of FreeRTOS that are supported by the device based on the tests passed.
- The SKU and the device name specified in the `device.json` file.
- The features of the device specified in the `device.json` file.
- The aggregate summary of test case results.
- A breakdown of test case results by libraries that were tested based on the device features (for example, FullWiFi, FullMQTT, and so on).
- Whether this qualification of FreeRTOS is for version 202012.00 that uses LTS libraries.

The `FRQ_Report.xml` is a report in standard JUnit XML format. You can integrate it into CI/CD platforms like Jenkins, Bamboo, and so on. The report contains the following elements:

- An aggregate summary of test case results.
- A breakdown of test case results by libraries that were tested based on the device features.

Interpret the IDT for FreeRTOS results

The report section in `awsiotdevicetester_report.xml` or `FRQ_Report.xml` lists the results of the tests that are executed.

The first XML tag `<testsuites>` contains the overall summary of the test execution. For example:

```
<testsuites name="FRQ results" time="5633" tests="184" failures="0"
errors="0" disabled="0">
```

Attributes used in the `<testsuites>` tag

`name`

The name of the test suite.

`time`

The time, in seconds, it took to run the qualification suite.

`tests`

The number of test cases executed.

`failures`

The number of test cases that were run, but did not pass.

`errors`

The number of test cases that IDT for FreeRTOS couldn't execute.

`disabled`

This attribute is not used and can be ignored.

If there are no test case failures or errors, your device meets the technical requirements to run FreeRTOS and can interoperate with AWS IoT services. If you choose to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the event of test case failures or errors, you can identify the test case that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag shows the test case result summary for a test group.

```
<testsuite name="FullMQTT" package="" tests="16" failures="0" time="76"
disabled="0" errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but with an attribute called `skipped` that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each of the test cases that were executed for a test group. For example:

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase"
attempts="1"></testcase>
```

Attributes used in the `<awsproduct>` tag

`name`

The name of the product being tested.

`version`

The version of the product being tested.

`sdk`

If you ran IDT with an SDK, this block contains the name and version of your SDK. If you didn't run IDT with an SDK, then this block contains:

```
<sdk>
    <name>N/A</vame>
    <version>N/A</version>
</sdk>
```

`features`

The features validated. Features marked as `required` are required to submit your board for qualification. The following snippet shows how this appears in the `awsiotdevicetester_report.xml` file.

```
<feature name="core-freertos" value="not-supported" type="required"></feature>
```

Features marked as `optional` are not required for qualification. The following snippets show optional features.

```
<feature name="ota-dataplane-mqtt" value="not-supported" type="optional"></feature>
<feature name="ota-dataplane-http" value="not-supported" type="optional"></feature>
```

If there are no test failures or errors for the required features, your device meets the technical requirements to run FreeRTOS and can interoperate with AWS IoT services. If you want to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the event of test failures or errors, you can identify the test that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag show the test result summary for a test group. For example:

```
<testsuite name="FreeRTOSVersion" package="" tests="1" failures="1" time="2"
 disabled="0" errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but has a `skipped` attribute that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each executed test for a test group. For example:

```
<testcase classname="FreeRTOSVersion" name="FreeRTOSVersion"></testcase>
```

`lts`

True if you are qualifying for a version of FreeRTOS that uses LTS libraries, false otherwise.

Attributes used in the `<testcase>` tag

`name`

The name of the test case.

`attempts`

The number of times IDT for FreeRTOS executed the test case.

When a test fails or an error occurs, `<failure>` or `<error>` tags are added to the `<testcase>` tag with information for troubleshooting. For example:

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase">
    <failure type="Failure">Reason for the test case failure</failure>
    <error>Reason for the test case execution error</error>
</testcase>
```

For more information, see Troubleshoot errors.

View the IDT for FreeRTOS logs

You can find logs that IDT for FreeRTOS generates from test execution in *devicetester-extract-location*/results/*execution-id*/logs. Two sets of logs are generated:

test_manager.log

Contains logs generated from IDT for FreeRTOS (for example, logs related configuration and report generation).

***test_group_id*__*test_case_id*.log (for example, FullMQTT__Full_MQTT.log)**

The log file for a test case, including output from the device under test. The log file is named according to the test group and test case that was run.

Develop and run your own IDT test suites

Starting in IDT v4.0.0, IDT for FreeRTOS combines a standardized configuration setup and result format with a test suite environment that enables you to develop custom test suites for your devices and device software. You can add custom tests for your own internal validation or provide them to your customers for device verification.

Use IDT to develop and run custom test suites, as follows:

To develop custom test suites

- Create test suites with custom test logic for the device that you want to test.
- Provide IDT with your custom test suites to test runners. Include information about specific settings configurations for your test suites.

To run custom test suites

- Set up the device that you want to test.
- Implement the settings configurations as required by the test suites that you want to use.
- Use IDT to run your custom test suites.
- View the test results and execution logs for the tests run by IDT.

Download latest version of AWS IoT Device Tester for FreeRTOS

Download the latest version of IDT and extract the software into a location on your file system where you have read and write permissions.

> **Note**
>
> IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.
> Windows has a path length limitation of 260 characters. If you are using Windows, extract IDT to a root directory like `C:\` or `D:\` to keep your paths under the 260 character limit.

Test suite workflow

Test suites are composed of three types of files:

- Configuration files that provide IDT with information on how to execute the test suite.
- Test executable files that IDT uses to run test cases.
- Additional files required to run tests.

Complete the following basic steps to create custom IDT tests:

1. Create configuration files for your test suite.
2. Create test case executables that contain the test logic for your test suite.
3. Verify and document the configuration information required for test runners to run the test suite.
4. Verify that IDT can run your test suite and produce test results as expected.

To quickly build a sample custom suite and run it, follow the instructions in Tutorial: Build and run the sample IDT test suite.

To get started creating a custom test suite in Python, see Tutorial: Develop a simple IDT test suite.

Tutorial: Build and run the sample IDT test suite

The AWS IoT Device Tester download includes the source code for a sample test suite. You can complete this tutorial to build and run the sample test suite to understand how you can use AWS IoT Device Tester for FreeRTOS to run custom test suites. Although this tutorial uses SSH, it is useful to learn how to use AWS IoT Device Tester with FreeRTOS devices.

In this tutorial, you will complete the following steps:

1. Build the sample test suite
2. Use IDT to run the sample test suite

Topics

- Set up the prerequisites for the sample test suite
- Configure device information for IDT
- Build the sample test suite
- Use IDT to run the sample test suite
- Troubleshoot errors

Set up the prerequisites for the sample test suite

To complete this tutorial, you need the following:

- **Host computer requirements**
  - Latest version of AWS IoT Device Tester
  - Python 3.7 or later

    To check the version of Python installed on your computer, run the following command:

    ```
    python3 --version
    ```

On Windows, if using this command returns an error, then use `python --version` instead. If the returned version number is 3.7 or greater, then run the following command in a Powershell terminal to set `python3` as an alias for your `python` command.

```
Set-Alias -Name "python3" -Value "python"
```

If no version information is returned or if the version number is less than 3.7, follow the instructions in Downloading Python to install Python 3.7+. For more information, see the Python documentation.

- urllib3

  To verify that `urllib3` is installed correctly, run the following command:

  ```
  python3 -c 'import urllib3'
  ```

  If `urllib3` is not installed, run the following command to install it:

  ```
  python3 -m pip install urllib3
  ```

- **Device requirements**
  - A device with a Linux operating system and a network connection to the same network as your host computer.

    We recommend that you use a Raspberry Pi with Raspberry Pi OS. Make sure you set up SSH on your Raspberry Pi to remotely connect to it.

Configure device information for IDT

Configure your device information for IDT to run the test. You must update the `device.json` template located in the *<device-tester-extract-location>*`/configs` folder with the following information.

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
```

```
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

In the `devices` object, provide the following information:

`id`

A user-defined unique identifier for your device.

`connectivity.ip`

The IP address of your device.

`connectivity.port`

Optional. The port number to use for SSH connections to your device.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`

- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.user`

The user name used to sign in to your device.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to your device.

This value applies only if `connectivity.auth.method` is set to `pki`.

`devices.connectivity.auth.credentials.password`

The password used for signing in to your device.

This value applies only if `connectivity.auth.method` is set to `password`.

> **Note**
>
> Specify `privKeyPath` only if `method` is set to `pki`.
> Specify `password` only if `method` is set to `password`.

Build the sample test suite

The *<device-tester-extract-location>*`/samples/python` folder contains sample configuration files, source code, and the IDT Client SDK that you can combine into a test suite using the provided build scripts. The following directory tree shows the location of these sample files:

```
<device-tester-extract-location>
### ...
### tests
### samples
#   ### ...
#   ### python
#       ### configuration
#       ### src
#       ### build-scripts
```

```
#           ### build.sh
#           ### build.ps1
### sdks
    ### ...
    ### python
        ### idt_client
```

To build the test suite, run the following commands on your host computer:

Windows

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.ps1
```

Linux, macOS, or UNIX

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.sh
```

This creates the sample test suite in the `IDTSampleSuitePython_1.0.0` folder within the *<device-tester-extract-location>*`/tests` folder. Review the files in the `IDTSampleSuitePython_1.0.0` folder to understand how the sample test suite is structured and to see various examples of test case executables and test configuration files.

> **Note**
>
> The sample test suite includes python source code. Do not include sensitive information in your test suite code.

Next step: Use IDT to run the sample test suite that you created.

Use IDT to run the sample test suite

To run the sample test suite, run the following commands on your host computer:

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id IDTSampleSuitePython
```

IDT runs the sample test suite and streams the results to the console. When the test has finished running, you see the following information:

```
========== Test Summary ==========
Execution Time:         5s
Tests Completed:        4
Tests Passed:           4
Tests Failed:           0
Tests Skipped:          0
---------------------------------
Test Groups:
    sample_group:       PASSED
---------------------------------
Path to AWS IoT Device Tester Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/IDTSampleSuitePython_Report.xml
```

Troubleshoot errors

Use the following information to help resolve any issues with completing the tutorial.

Test case does not run successfully

- If the test does not run successfully, IDT streams the error logs to the console that can help you troubleshoot the test run. Make sure that you meet all the prerequisites for this tutorial.

Cannot connect to the device under test

Verify the following:

- Your `device.json` file contains the correct IP address, port, and authentication information.
- You can connect to your device over SSH from your host computer.

Tutorial: Develop a simple IDT test suite

A test suite combines the following:

- Test executable that contain the test logic

- Configuration files that describe the test suite

This tutorial shows you how to use IDT for FreeRTOS to develop a Python test suite that contains a single test case. Although this tutorial uses SSH, it is useful to learn how to use AWS IoT Device Tester with FreeRTOS devices.

In this tutorial, you will complete the following steps:

1. Create a test suite directory
2. Create configuration files
3. Create the test case executable
4. Run the test suite

Follow the steps below to complete a tutorial for developing a simple IDT test suite.

Topics

- Set up the prerequisites for a simple IDT test suite
- Create a test suite directory
- Create configuration files
- Get the IDT client SDK
- Create the test case executable
- Configure device information for IDT
- Run the test suite
- Troubleshoot errors
- Create IDT test suite configuration files
- Configure the IDT test orchestrator
- Configure the IDT state machine
- Create IDT test case executable
- Use the IDT context
- Configure settings for test runners
- Debug and run custom test suites
- Review IDT test results and logs
- Submit IDT usage metrics

Set up the prerequisites for a simple IDT test suite

To complete this tutorial, you need the following:

- **Host computer requirements**
  - Latest version of AWS IoT Device Tester
  - Python 3.7 or later

    To check the version of Python installed on your computer, run the following command:

    ```
    python3 --version
    ```

    On Windows, if using this command returns an error, then use `python --version` instead. If the returned version number is 3.7 or greater, then run the following command in a Powershell terminal to set `python3` as an alias for your `python` command.

    ```
    Set-Alias -Name "python3" -Value "python"
    ```

    If no version information is returned or if the version number is less than 3.7, follow the instructions in Downloading Python to install Python 3.7+. For more information, see the Python documentation.
  - urllib3

    To verify that `urllib3` is installed correctly, run the following command:

    ```
    python3 -c 'import urllib3'
    ```

    If `urllib3` is not installed, run the following command to install it:

    ```
    python3 -m pip install urllib3
    ```
- **Device requirements**
  - A device with a Linux operating system and a network connection to the same network as your host computer.

    We recommend that you use a Raspberry Pi with Raspberry Pi OS. Make sure you set up SSH on your Raspberry Pi to remotely connect to it.

Create a test suite directory

IDT logically separates test cases into test groups within each test suite. Each test case must be inside a test group. For this tutorial, create a folder called `MyTestSuite_1.0.0` and create the following directory tree within this folder:

```
MyTestSuite_1.0.0
### suite
    ### myTestGroup
        ### myTestCase
```

Create configuration files

Your test suite must contain the following required configuration files:

Required files

`suite.json`

Contains information about the test suite. See Configure suite.json.

`group.json`

Contains information about a test group. You must create a `group.json` file for each test group in your test suite. See Configure group.json.

`test.json`

Contains information about a test case. You must create a `test.json` file for each test case in your test suite. See Configure test.json.

1. In the `MyTestSuite_1.0.0/suite` folder, create a `suite.json` file with the following structure:

   ```
   {
       "id": "MyTestSuite_1.0.0",
       "title": "My Test Suite",
       "details": "This is my test suite.",
       "userDataRequired": false
   }
   ```

2. In the `MyTestSuite_1.0.0/myTestGroup` folder, create a `group.json` file with the following structure:

```
{
    "id": "MyTestGroup",
    "title": "My Test Group",
    "details": "This is my test group.",
    "optional": false
}
```

3. In the `MyTestSuite_1.0.0/myTestGroup/myTestCase` folder, create a `test.json` file with the following structure:

```
{
    "id": "MyTestCase",
    "title": "My Test Case",
    "details": "This is my test case.",
    "execution": {
        "timeout": 300000,
        "linux": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "mac": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "win": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        }
    }
}
```

The directory tree for your `MyTestSuite_1.0.0` folder should now look like the following:

```
MyTestSuite_1.0.0
### suite
```

```
    ### suite.json
    ### myTestGroup
        ### group.json
        ### myTestCase
            ### test.json
```

Get the IDT client SDK

You use the IDT client SDK to enable IDT to interact with the device under test and to report test results. For this tutorial, you will use the Python version of the SDK.

From the *<device-tester-extract-location>*/sdks/python/ folder, copy the `idt_client` folder to your `MyTestSuite_1.0.0/suite/myTestGroup/myTestCase` folder.

To verify that the SDK was successfully copied, run the following command.

```
cd MyTestSuite_1.0.0/suite/myTestGroup/myTestCase
python3 -c 'import idt_client'
```

Create the test case executable

Test case executables contain the test logic that you want to run. A test suite can contain multiple test case executables. For this tutorial, you will create only one test case executable.

1. Create the test suite file.

   In the `MyTestSuite_1.0.0/suite/myTestGroup/myTestCase` folder, create a `myTestCase.py` file with the following content:

   ```
   from idt_client import *

   def main():
       # Use the client SDK to communicate with IDT
       client = Client()

   if __name__ == "__main__":
       main()
   ```

2. Use client SDK functions to add the following test logic to your `myTestCase.py` file:

   a. Run an SSH command on the device under test.

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

if __name__ == "__main__":
    main()
```

b. Send the test result to IDT.

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

    # Create a send result request
    sr_req = SendResultRequest(TestResult(passed=True))

    # Send the result
    client.send_result(sr_req)
```

```
if __name__ == "__main__":
    main()
```

Configure device information for IDT

Configure your device information for IDT to run the test. You must update the `device.json` template located in the *<device-tester-extract-location>*`/configs` folder with the following information.

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

In the `devices` object, provide the following information:

`id`

A user-defined unique identifier for your device.

`connectivity.ip`

The IP address of your device.

`connectivity.port`

Optional. The port number to use for SSH connections to your device.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.user`

The user name used to sign in to your device.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to your device.

This value applies only if `connectivity.auth.method` is set to `pki`.

`devices.connectivity.auth.credentials.password`

The password used for signing in to your device.

This value applies only if `connectivity.auth.method` is set to `password`.

Note

Specify `privKeyPath` only if `method` is set to `pki`.

Specify `password` only if `method` is set to `password`.

Run the test suite

After you create your test suite, you want to make sure that it functions as expected. Complete the following steps to run the test suite with your existing device pool to do so.

1. Copy your `MyTestSuite_1.0.0` folder into *<device-tester-extract-location>*`/tests`.
2. Run the following commands:

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id MyTestSuite
```

IDT runs your test suite and streams the results to the console. When the test has finished running, you see the following information:

```
time="2020-10-19T09:24:47-07:00" level=info msg=Using pool: pool
time="2020-10-19T09:24:47-07:00" level=info msg=Using test suite "MyTestSuite_1.0.0"
 for execution
time="2020-10-19T09:24:47-07:00" level=info msg=b'hello world\n'
 suiteId=MyTestSuite groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:47-07:00" level=info msg=All tests finished.
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:48-07:00" level=info msg=

========== Test Summary ==========
Execution Time:         1s
Tests Completed:        1
Tests Passed:           1
Tests Failed:           0
Tests Skipped:          0
---------------------------------
Test Groups:
    myTestGroup:        PASSED
---------------------------------
Path to AWS IoT Device Tester Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/awsiotdevicetester_report.xml
```

```
Path to Test Execution Logs: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/MyTestSuite_Report.xml
```

Troubleshoot errors

Use the following information to help resolve any issues with completing the tutorial.

Test case does not run successfully

If the test does not run successfully, IDT streams the error logs to the console that can help you troubleshoot the test run. Before you check the error logs, verify the following:

- The IDT client SDK is in the correct folder as described in Get the IDT client SDK.
- You meet all the prerequisites for this tutorial. For more information, see Set up the prerequisites for a simple IDT test suite.

Cannot connect to the device under test

Verify the following:

- Your `device.json` file contains the correct IP address, port, and authentication information.
- You can connect to your device over SSH from your host computer.

Create IDT test suite configuration files

This section describes the formats in which you create configuration files that you include when you write a custom test suite.

Required configuration files

`suite.json`

Contains information about the test suite. See Configure suite.json.

`group.json`

Contains information about a test group. You must create a `group.json` file for each test group in your test suite. See Configure group.json.

`test.json`

Contains information about a test case. You must create a `test.json` file for each test case in your test suite. See Configure test.json.

Optional configuration files

`test_orchestrator.yaml` or `state_machine.json`

Defines how tests are run when IDT runs the test suite. SSe Configure test_orchestrator.yaml.

> **Note**
>
> Starting in IDT v4.5.2, you use the `test_orchestrator.yaml` file to define the test workflow. In previous versions of IDT, you use the `state_machine.json` file. For information about the state machine, see Configure the IDT state machine.

`userdata_schema.json`

Defines the schema for the `userdata.json` file that test runners can include in their setting configuration. The `userdata.json` file is used for any additional configuration information that is required to run the test but is not present in the `device.json` file. See Configure userdata_schema.json.

Configuration files are placed in your *<custom-test-suite-folder>* as shown here.

```
<custom-test-suite-folder>
### suite
    ### suite.json
    ### test_orchestrator.yaml
    ### userdata_schema.json
    ### <test-group-folder>
        ### group.json
        ### <test-case-folder>
            ### test.json
```

Configure suite.json

The `suite.json` file sets environment variables and determines whether user data is required to run the test suite. Use the following template to configure your *<custom-test-suite-folder>*`/suite/suite.json` file:

```
{
    "id": "<suite-name>_<suite-version>",
    "title": "<suite-title>",
    "details": "<suite-details>",
    "userDataRequired": true | false,
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        },
        ...
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

All fields that contain values are required as described here:

`id`

A unique user-defined ID for the test suite. The value of `id` must match the name of the test suite folder in which the `suite.json` file is located. The suite name and suite version must also meet the following requirements:

- *<suite-name>* cannot contain underscores.
- *<suite-version>* is denoted as *x.x.x*, where x is a number.

The ID is shown in IDT-generated test reports.

`title`

A user-defined name for the product or feature being tested by this test suite. The name is displayed in the IDT CLI for test runners.

`details`

A short description of the purpose of the test suite.

`userDataRequired`

Defines whether test runners need to include custom information in a `userdata.json` file. If you set this value to `true`, you must also include the `userdata_schema.json` file in your test suite folder.

`environmentVariables`

Optional. An array of environment variables to set for this test suite.

`environmentVariables.key`

The name of the environment variable.

`environmentVariables.value`

The value of the environment variable.

Configure group.json

The `group.json` file defines whether a test group is required or optional. Use the following template to configure your *`<custom-test-suite-folder>`*`/suite/`*`<test-group>`*`/group.json` file:

```
{
    "id": "<group-id>",
    "title": "<group-title>",
    "details": "<group-details>",
    "optional": true | false,
}
```

All fields that contain values are required as described here:

`id`

A unique user-defined ID for the test group. The value of `id` must match the name of the test group folder in which the `group.json` file is located and should not have underscores (_). The ID is used in IDT-generated test reports.

`title`

A descriptive name for the test group. The name is displayed in the IDT CLI for test runners.

`details`

A short description of the purpose of the test group.

`optional`

Optional. Set to `true` to display this test group as an optional group after IDT finishes running required tests. Default value is `false`.

Configure test.json

The `test.json` file determines the test case executables and the environment variables that are used by a test case. For more information about creating test case executables, see Create IDT test case executable.

Use the following template to configure your *<custom-test-suite-folder>*/suite/*<test-group>*/*<test-case>*/test.json file:

```
{
    "id": "<test-id>",
    "title": "<test-title>",
    "details": "<test-details>",
    "requireDUT": true | false,
    "requiredResources": [
        {
            "name": "<resource-name>",
            "features": [
                {
                    "name": "<feature-name>",
                    "version": "<feature-version>",
                    "jobSlots": <job-slots>
                }
            ]
        }
    ],
    "execution": {
        "timeout": <timeout>,
        "mac": {
            "cmd": "/path/to/executable",
            "args": [
```

```
                "<argument>"
            ],
        },
        "linux": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "win": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ]
        }
    },
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

All fields that contain values are required as described here:

`id`

A unique user-defined ID for the test case. The value of `id` must match the name of the test case folder in which the `test.json` file is located and should not have underscores (_). The ID is used in IDT-generated test reports.

`title`

A descriptive name for the test case. The name is displayed in the IDT CLI for test runners.

`details`

A short description of the purpose of the test case.

`requireDUT`

Optional. Set to `true` if a device is required to run this test, otherwise set to `false`. Default value is `true`. Test runners will configure the devices they will use to run the test in their `device.json` file.

`requiredResources`

Optional. An array that provides information about resource devices needed to run this test.

`requiredResources.name`

The unique name to give the resource device when this test is running.

`requiredResources.features`

An array of user-defined resource device features.

`requiredResources.features.name`

The name of the feature. The device feature for which you want to use this device. This name is matched against the feature name provided by the test runner in the `resource.json` file.

`requiredResources.features.version`

Optional. The version of the feature. This value is matched against the feature version provided by the test runner in the `resource.json` file. If a version is not provided, then the feature is not checked. If a version number is not required for the feature, leave this field blank.

`requiredResources.features.jobSlots`

Optional. The number of simultaneous tests that this feature can support. The default value is 1. If you want IDT to use distinct devices for individual features, then we recommend that you set this value to 1.

`execution.timeout`

The amount of time (in milliseconds) that IDT waits for the test to finish running. For more information about setting this value, see Create IDT test case executable.

`execution.os`

The test case executables to run based on the operating system of the host computer that runs IDT. Supported values are `linux`, `mac`, and `win`.

`execution.os.cmd`

The path to the test case executable that you want to run for the specified operating system. This location must be in the system path.

`execution.os.args`

Optional. The arguments to provide to run the test case executable.

`environmentVariables`

Optional. An array of environment variables set for this test case.

`environmentVariables.key`

The name of the environment variable.

`environmentVariables.value`

The value of the environment variable.

> **Note**
>
> If you specify the same environment variable in the `test.json` file and in the `suite.json` file, the value in the `test.json` file takes precedence.

Configure test_orchestrator.yaml

A test orchestrator is a construct that controls the test suite execution flow. It determines the starting state of a test suite, manages state transitions based on user-defined rules, and continues to transition through those states until it reaches the end state.

If your test suite doesn't include a user-defined test orchestrator, IDT will generate a test orchestrator for you.

The default test orchestrator performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

For more information about how the IDT test orchestrator functions, see Configure the IDT test orchestrator.

Configure userdata_schema.json

The `userdata_schema.json` file determines the schema in which test runners provide user data. User data is required if your test suite requires information that is not present in the `device.json` file. For example, your tests might need Wi-Fi network credentials, specific open ports, or certificates that a user must provide. This information can be provided to IDT as an input parameter called `userdata`, the value for which is a `userdata.json` file, that users create in their *<device-tester-extract-location>*`/config` folder. The format of the `userdata.json` file is based on the `userdata_schema.json` file that you include in the test suite.

To indicate that test runners must provide a `userdata.json` file:

1. In the `suite.json` file, set `userDataRequired` to `true`.
2. In your *<custom-test-suite-folder>*, create a `userdata_schema.json` file.
3. Edit the `userdata_schema.json` file to create a valid IETF Draft v4 JSON Schema.

When IDT runs your test suite, it automatically reads the schema and uses it to validate the `userdata.json` file provided by the test runner. If valid, the contents of the `userdata.json` file are available in both the IDT context and in the test orchestrator context.

Configure the IDT test orchestrator

Starting in IDT v4.5.2, IDT includes a new *test orchestrator* component. The test orchestrator is an IDT component that controls the test suite execution flow, and generates the test report after IDT finishes running all tests. The test orchestrator determines test selection and the order in which tests are run based on user-defined rules.

If your test suite doesn't include a user-defined test orchestrator, IDT will generate a test orchestrator for you.

The default test orchestrator performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

The test orchestrator replaces the IDT state machine. We strongly recommend that you use the test orchestrator to develop your test suites instead of the IDT state machine. The test orchestrator provides the following improved features:

- Uses a declarative format compared to the imperative format that the IDT state machine uses. This allows you to specify which tests you want to run and when you want to run them.
- Manages specific group handling, report generation, error handling, and result tracking so that you aren't required to manually manage these actions.
- Uses the YAML format, which supports comments by default.
- Requires 80 percent less disk space than the test orchestrator to define the same workflow.
- Adds pre-test validation to verify that your workflow definition doesn't contain incorrect test IDs or circular dependencies.

Test orchestrator format

You can use the following template to configure your own *custom-test-suite-folder*/suite/test_orchestrator.yaml file:

```
Aliases:
  string: context-expression

ConditionalTests:
  - Condition: context-expression
    Tests:
      - test-descriptor

Order:
  - - group-descriptor
    - group-descriptor

Features:
  - Name: feature-name
    Value: support-description
    Condition: context-expression
    Tests:
        - test-descriptor
    OneOfTests:
        - test-descriptor
    IsRequired: boolean
```

All fields that contain values are required as described here:

`Aliases`

Optional. User-defined strings that map to context expressions. Aliases allow you to generate friendly names to identify context expressions in your test orchestrator configuration. This is especially useful if you're creating complex context expressions or expressions that you use in multiple places.

You can use context expressions to store context queries that allow you to access data from other IDT configurations. For more information, see Access data in the context.

Example

Example

```
Aliases:
    FizzChosen: "'{{$pool.features[?(@.name == 'Fizz')].value[0]}}' == 'yes'"
    BuzzChosen: "'{{$pool.features[?(@.name == 'Buzz')].value[0]}}' == 'yes'"
    FizzBuzzChosen: "'{{$aliases.FizzChosen}}' && '{{$aliases.BuzzChosen}}'"
```

`ConditionalTests`

Optional. A list of conditions, and the corresponding test cases that are run when each condition is satisfied. Each condition can have multiple test cases; however, you can assign a given test case to only one condition.

By default, IDT runs any test case that isn't assigned to a condition in this list. If you don't specify this section, IDT runs all test groups in the test suite.

Each item in the `ConditionalTests` list includes the following parameters:

`Condition`

A context expression that evaluates to a Boolean value. If the evaluated value is true, IDT runs the test cases that are specified in the `Tests` parameter.

`Tests`

The list of test descriptors.

Each test descriptor uses the test group ID and one or more test case IDs to identify the individual tests to run from a specific test group. The test descriptor uses the following format:

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

Example

Example

The following example uses generic context expressions that you can define as `Aliases`.

```
ConditionalTests:
    - Condition: "{{$aliases.Condition1}}"
      Tests:
          - GroupId: A
          - GroupId: B
    - Condition: "{{$aliases.Condition2}}"
      Tests:
          - GroupId: D
    - Condition: "{{$aliases.Condition1}} || {{$aliases.Condition2}}"
      Tests:
          - GroupId: C
```

Based on the defined conditions, IDT selects test groups as follows:

- If `Condition1` is true, IDT runs the tests in test groups A, B, and C.
- If `Condition2` is true, IDT runs the tests in test groups C and D.

`Order`

Optional. The order in which to run tests. You specify the test order at the test group level. If you don't specify this section, IDT runs all applicable test groups in a random order. The value of `Order` is a list of group descriptor lists. Any test group that you don't list in `Order`, can be run in parallel with any other listed test group.

Each group descriptor list contains one of more group descriptors, and identifies the order in which to run the groups that are specified in each descriptor. You can use the following formats to define individual group descriptors:

- *`group-id`*—The group ID of an existing test group.
- [*`group-id`*, *`group-id`*]—List of test groups that can be run in any order relative to each other.

- "*"—Wildcard. This is equivalent to the list of all test groups that are not already specified in the current group descriptor list.

The value for `Order` must also meet the following requirements:

- Test group IDs that you specify in a group descriptor must exist in your test suite.
- Each group descriptor list must include at least one test group.
- Each group descriptor list must contain unique group IDs. You cannot repeat a test group ID within individual group descriptors.
- A group descriptor list can have at most one wildcard group descriptor. The wildcard group descriptor must be the first or the last item in the list.

Example

Example

For a test suite that contains test groups A, B, C, D, and E, the following list of examples shows different ways to specify that IDT should first run test group A, then run test group B, and then run test groups C, D, and E in any order.

- ```
  Order:
      - - A
        - B
        - [C, D, E]
  ```
- ```
  Order:
      - - A
        - B
        - "*"
  ```
- ```
  Order:
      - - A
        - B

      - - B
        - C

      - - B
        - D

      - - B
  ```
  ```

```
      - E
```

`Features`

Optional. The list of product features that you want IDT to add to the `awsiotdevicetester_report.xml` file. If you don't specify this section, IDT won't add any product features to the report.

A product feature is user-defined information about specific criteria that a device might meet. For example, the MQTT product feature can designate that the device publishes MQTT messages properly. In `awsiotdevicetester_report.xml`, product features are set as `supported`, `not-supported`, or a custom user-defined value, based on whether specified tests passed.

Each item in the `Features` list consists of the following parameters:

`Name`

The name of the feature.

`Value`

Optional. The custom value that you want to use in the report instead of `supported`. If this value is not specified, then based IDT sets the feature value to `supported` or `not-supported` based on test results. If you test the same feature with different conditions, you can use a custom value for each instance of that feature in the `Features` list, and IDT concatenates the feature values for supported conditions. For more information, see

`Condition`

A context expression that evaluates to a Boolean value. If the evaluated value is true, IDT adds the feature to the test report after it finishes running the test suite. If the evaluated value is false, the test is not included in the report.

`Tests`

Optional. The list of test descriptors. All of the tests that are specified in this list must pass for the feature to be supported.

Each test descriptor in this list uses the test group ID and one or more test case IDs to identify the individual tests to run from a specific test group. The test descriptor uses the following format:

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

You must specify either `Tests` or `OneOfTests` for each feature in the `Features` list.

`OneOfTests`

Optional. The list of test descriptors. At least one of the tests that are specified in this list must pass for the feature to be supported.

Each test descriptor in this list uses the test group ID and one or more test case IDs to identify the individual tests to run from a specific test group. The test descriptor uses the following format:

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

You must specify either `Tests` or `OneOfTests` for each feature in the `Features` list.

`IsRequired`

The boolean value that defines whether the feature is required in the test report. The default value is `false`.

Test orchestrator context

The test orchestrator context is a read-only JSON document that contains data that is available to the test orchestrator during execution. The test orchestrator context is accessible only from the test orchestrator, and contains information that determines the test flow. For example, you can use information configured by test runners in the `userdata.json` file to determine whether a specific test is required to run.

The test orchestrator context uses the following format:

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
```

```
    "config": {
        <config-json-content>
    }
}
```

`pool`

Information about the device pool selected for the test run. For a selected device pool, this information is retrieved from the corresponding top-level device pool array element defined in the `device.json` file.

`userData`

Information in the `userdata.json` file.

`config`

Information in the `config.json` file.

You can query the context using JSONPath notation. The syntax for JSONPath queries in state definitions is `{{query}}`. When you access data from the test orchestrator context, make sure that each value evaluates to a string, a number, or a Boolean.

For more information about using JSONPath notation to access data from the context, see Use the IDT context.

Configure the IDT state machine

Important

Starting in IDT v4.5.2, this state machine is deprecated. We strongly recommend that you use the new test orchestrator. For more information, see Configure the IDT test orchestrator.

A state machine is a construct that controls the test suite execution flow. It determines the starting state of a test suite, manages state transitions based on user-defined rules, and continues to transition through those states until it reaches the end state.

If your test suite doesn't include a user-defined state machine, IDT will generate a state machine for you. The default state machine performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

The state machine for an IDT test suite must meet the following criteria:

- Each state corresponds to an action for IDT to take, such as to run a test group or product a report file.
- Transitioning to a state executes the action associated with the state.
- Each state defines the transition rule for the next state.
- The end state must be either `Succeed` or `Fail`.

State machine format

You can use the following template to configure your own *<custom-test-suite-folder>*`/suite/state_machine.json` file:

```
{
  "Comment": "<description>",
  "StartAt": "<state-name>",
  "States": {
    "<state-name>": {
      "Type": "<state-type>",
      // Additional state configuration
    }

    // Required states
    "Succeed": {
      "Type": "Succeed"
    },
    "Fail": {
      "Type": "Fail"
    }
  }
}
```

All fields that contain values are required as described here:

`Comment`

A description of the state machine.

`StartAt`

The name of the state at which IDT starts running the test suite. The value of `StartAt` must be set to one of the states listed in the `States` object.

`States`

An object that maps user-defined state names to valid IDT states. Each States.*`state-name`* object contains the definition of a valid state mapped to the *`state-name`*.

The `States` object must include the `Succeed` and `Fail` states. For information about valid states, see Valid states and state definitions.

Valid states and state definitions

This section describes the state definitions of all of the valid states that can be used in the IDT state machine. Some of the following states support configurations at the test case level. However, we recommend that you configure state transition rules at the test group level instead of the test case level unless absolutely necessary.

State definitions

- RunTask
- Choice
- Parallel
- AddProductFeatures
- Report
- LogMessage
- SelectGroup
- Fail
- Succeed

RunTask

The `RunTask` state runs test cases from a test group defined in the test suite.

```
{
    "Type": "RunTask",
    "Next": "<state-name>",
    "TestGroup": "<group-id>",
    "TestCases": [
        "<test-id>"
    ],
    "ResultVar": "<result-name>"
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`TestGroup`

Optional. The ID of the test group to run. If this value is not specified, then IDT runs the test group that the test runner selects.

`TestCases`

Optional. An array of test case IDs from the group specified in `TestGroup`. Based on the values of `TestGroup` and `TestCases`, IDT determines the test execution behavior as follows:

- When both `TestGroup` and `TestCases` are specified, IDT runs the specified test cases from the test group.
- When `TestCases` are specified but `TestGroup` is not specified, IDT runs the specified test cases.
- When `TestGroup` is specified, but `TestCases` is not specified, IDT runs all of the test cases within the specified test group.
- When neither `TestGroup` or `TestCases` is specified, IDT runs all test cases from the test group that the test runner selects from the IDT CLI. To enable group selection for test runners, you must include both `RunTask` and `Choice` states in your `statemachine.json` file. For an example of how this works, see Example state machine: Run user-selected test groups.

  For more information about enabling IDT CLI commands for test runners, see the section called “Enable IDT CLI commands”.

`ResultVar`

The name of the context variable to set with the results of the test run. Do not specify this value if you did not specify a value for `TestGroup`. IDT sets the value of the variable that you define in `ResultVar` to `true` or `false` based on the following:

- If the variable name is of the form *`text`*`_`*`text`*`_passed`, then the value is set to whether all tests in the first test group passed or were skipped.
- In all other cases, the value is set to whether all tests in all test groups passed or were skipped.

Typically, you will use `RunTask` state to specify a test group ID without specifying individual test case IDs, so that IDT will run all of the test cases in the specified test group. All test cases that are run by this state run in parallel, in a random order. However, if all of the test cases require a device to run, and only a single device is available, then the test cases will run sequentially instead.

Error handling

If any of the specified test groups or test case IDs are not valid, then this state issues the `RunTaskError` execution error. If the state encounters an execution error, then it also sets the `hasExecutionError` variable in the state machine context to `true`.

Choice

The `Choice` state lets you dynamically set the next state to transition to based on user-defined conditions.

```
{
    "Type": "Choice",
    "Default": "<state-name>",
    "FallthroughOnError": true | false,
    "Choices": [
        {
            "Expression": "<expression>",
            "Next": "<state-name>"
        }
    ]
}
```

All fields that contain values are required as described here:

`Default`

The default state to transition to if none of the expressions defined in `Choices` can be evaluated to `true`.

`FallthroughOnError`

Optional. Specifies the behavior when the state encounters an error in evaluating expressions. Set to `true` if you want to skip an expression if the evaluation results in an error. If no expressions match, then the state machine transitions to the `Default` state. If the `FallthroughOnError` value is not specified, it defaults to `false`.

`Choices`

An array of expressions and states to determine which state to transition to after executing the actions in the current state.

`Choices.Expression`

An expression string that evaluates to a boolean value. If the expression evaluates to `true`, then the state machine transitions to the state defined in `Choices.Next`. Expression strings retrieve values from the state machine context and then perform operations on them to arrive at a boolean value. For information about accessing the state machine context, see State machine context.

`Choices.Next`

The name of the state to transition to if the expression defined in `Choices.Expression` evaluates to `true`.

Error handling

The `Choice` state can require error handling in the following cases:

- Some variables in the choice expressions don't exist in the state machine context.
- The result of an expression is not a boolean value.
- The result of a JSON lookup is not a string, number, or boolean.

You cannot use a `Catch` block to handle errors in this state. If you want to stop executing the state machine when it encounters an error, you must set `FallthroughOnError` to `false`. However, we

recommend that you set `FallthroughOnError` to `true`, and depending on your use case, do one of the following:

- If a variable you are accessing is expected to not exist in some cases, then use the value of `Default` and additional `Choices` blocks to specify the next state.
- If a variable that you are accessing should always exist, then set the `Default` state to `Fail`.

Parallel

The `Parallel` state lets you define and run new state machines in parallel with each other.

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Branches": [
        <state-machine-definition>
    ]
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`Branches`

An array of state machine definitions to run. Each state machine definition must contain its own `StartAt`, `Succeed`, and `Fail` states. The state machine definitions in this array cannot reference states outside of their own definition.

> **Note**
>
> Because each branch state machine shares the same state machine context, setting variables in one branch and then reading those variables from another branch might result in unexpected behavior.

The `Parallel` state moves to the next state only after it runs all of the branch state machines. Each state that requires a device will wait to run until the device is available. If multiple devices

are available, this state runs test cases from multiple groups in parallel. If enough devices are not available, then test cases will run sequentially. Because test cases are run in a random order when they run in parallel, different devices might be used to run tests from the same test group.

Error handling

Make sure that both the branch state machine and the parent state machine transition to the `Fail` state to handle execution errors.

Because branch state machines do not transmit execution errors to the parent state machine, you cannot use a `Catch` block to handle execution errors in branch state machines. Instead, use the `hasExecutionErrors` value in the shared state machine context. For an example of how this works, see Example state machine: Run two test groups in parallel.

AddProductFeatures

The `AddProductFeatures` state lets you add product features to the `awsiotdevicetester_report.xml` file generated by IDT.

A product feature is user-defined information about specific criteria that a device might meet. For example, the `MQTT` product feature can designate that the device publishes MQTT messages properly. In the report, product features are set as `supported`, `not-supported`, or a custom value, based on whether specified tests passed.

> **Note**
>
> The `AddProductFeatures` state does not generate reports by itself. This state must transition to the `Report` state to generate reports.

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Features": [
        {
            "Feature": "<feature-name>",
            "Groups": [
                "<group-id>"
            ],
```

```
            "OneOfGroups": [
                "<group-id>"
            ],
            "TestCases": [
                "<test-id>"
            ],
            "IsRequired": true | false,
            "ExecutionMethods": [
                "<execution-method>"
            ]
        }
    ]
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`Features`

An array of product features to show in the `awsiotdevicetester_report.xml` file.

`Feature`

The name of the feature

`FeatureValue`

Optional. The custom value to use in the report instead of `supported`. If this value is not specified, then based on test results, the feature value is set to `supported` or `not-supported`.

If you use a custom value for `FeatureValue`, you can test the same feature with different conditions, and IDT concatenates the feature values for the supported conditions. For example, the following excerpt shows the `MyFeature` feature with two separate feature values:

```
...
{
    "Feature": "MyFeature",
    "FeatureValue": "first-feature-supported",
    "Groups": ["first-feature-group"]
```

```
},
{
    "Feature": "MyFeature",
    "FeatureValue": "second-feature-supported",
    "Groups": ["second-feature-group"]
},
...
```

If both test groups pass, then the feature value is set to `first-feature-supported, second-feature-supported`.

`Groups`

Optional. An array of test group IDs. All tests within each specified test group must pass for the feature to be supported.

`OneOfGroups`

Optional. An array of test group IDs. All tests within at least one of the specified test groups must pass for the feature to be supported.

`TestCases`

Optional. An array of test case IDs. If you specify this value, then the following apply:

- All of the specified test cases must pass for the feature to be supported.
- `Groups` must contain only one test group ID.
- `OneOfGroups` must not be specified.

`IsRequired`

Optional. Set to `false` to mark this feature as an optional feature in the report. The default value is `true`.

`ExecutionMethods`

Optional. An array of execution methods that match the `protocol` value specified in the `device.json` file. If this value is specified, then test runners must specify a `protocol` value that matches one of the values in this array to include the feature in the report. If this value is not specified, then the feature will always be included in the report.

To use the `AddProductFeatures` state, you must set the value of `ResultVar` in the `RunTask` state to one of the following values:

- If you specified individual test case IDs, then set `ResultVar` to *`group-id_test-id`*`_passed`.
- If you did not specify individual test case IDs, then set `ResultVar` to *`group-id`*`_passed`.

The `AddProductFeatures` state checks for test results in the following manner:

- If you did not specify any test case IDs, then the result for each test group is determined from the value of the *`group-id`*`_passed` variable in the state machine context.
- If you did specify test case IDs, then the result for each of the tests is determined from the value of the *`group-id_test-id`*`_passed` variable in the state machine context.

Error handling

If a group ID provided in this state is not a valid group ID, then this state results in the `AddProductFeaturesError` execution error. If the state encounters an execution error, then it also sets the `hasExecutionErrors` variable in the state machine context to `true`.

Report

The `Report` state generates the *`suite-name`*`_Report.xml` and `awsiotdevicetester_report.xml` files. This state also streams the report to the console.

```
{
    "Type": "Report",
    "Next": "<state-name>"
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

You should always transition to the `Report` state towards the end of the test execution flow so that test runners can view test results. Typically, the next state after this state is `Succeed`.

Error handling

If this state encounters issues with generating the reports, then it issues the `ReportError` execution error.

LogMessage

The `LogMessage` state generates the `test_manager.log` file and streams the log message to the console.

```
{
    "Type": "LogMessage",
    "Next": "<state-name>"
    "Level": "info | warn | error"
    "Message": "<message>"
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`Level`

The error level at which to create the log message. If you specify a level that is not valid, this state generates an error message and discards it.

`Message`

The message to log.

SelectGroup

The `SelectGroup` state updates the state machine context to indicate which groups are selected. The values set by this state are used by any subsequent `Choice` states.

```
{
    "Type": "SelectGroup",
    "Next": "<state-name>"
    "TestGroups": [
        <group-id>"
    ]
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`TestGroups`

An array of test groups that will be marked as selected. For each test group ID in this array, the *`group-id`*`_selected` variable is set to `true` in the context. Make sure that you provide valid test group IDs because IDT does not validate whether the specified groups exist.

Fail

The `Fail` state indicates that the state machine did not execute correctly. This is an end state for the state machine, and each state machine definition must include this state.

```
{
    "Type": "Fail"
}
```

Succeed

The `Succeed` state indicates that the state machine executed correctly. This is an end state for the state machine, and each state machine definition must include this state.

```
{
    "Type": "Succeed"
}
```

State machine context

The state machine context is a read-only JSON document that contains data that is available to the state machine during execution. The state machine context is accessible only from the state machine, and contains information that determines the test flow. For example, you can use information configured by test runners in the `userdata.json` file to determine whether a specific test is required to run.

The state machine context uses the following format:

```
{
    "pool": {
        <device-json-pool-element>
```

```
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    },
    "suiteFailed": true | false,
    "specificTestGroups": [
        "<group-id>"
    ],
    "specificTestCases": [
        "<test-id>"
    ],
    "hasExecutionErrors": true
}
```

`pool`

Information about the device pool selected for the test run. For a selected device pool, this information is retrieved from the corresponding top-level device pool array element defined in the `device.json` file.

`userData`

Information in the `userdata.json` file.

`config`

Information pin the `config.json` file.

`suiteFailed`

The value is set to `false` when the state machine starts. If a test group fails in a `RunTask` state, then this value is set to `true` for the remaining duration of the state machine execution.

`specificTestGroups`

If the test runner selects specific test groups to run instead of the entire test suite, this key is created and contains the list of specific test group IDs.

`specificTestCases`

If the test runner selects specific test cases to run instead of the entire test suite, this key is created and contains the list of specific test case IDs.

`hasExecutionErrors`

Does not exit when the state machine starts. If any state encounters an execution errors, this variable is created and set to `true` for the remaining duration of the state machine execution.

You can query the context using JSONPath notation. The syntax for JSONPath queries in state definitions is `{{$.query}}`. You can use JSONPath queries as placeholder strings within some states. IDT replaces the placeholder strings with the value of the evaluated JSONPath query from the context. You can use placeholders for the following values:

- The `TestCases` value in `RunTask` states.
- The `Expression` value `Choice` state.

When you access data from the state machine context, make sure the following conditions are met:

- Your JSON paths must begin with `$.`
- Each value must evaluate to a string, a number, or a boolean.

For more information about using JSONPath notation to access data from the context, see Use the IDT context.

Execution errors

Execution errors are errors in the state machine definition that the state machine encounters when executing a state. IDT logs information about each error in the `test_manager.log` file and streams the log message to the console.

You can use the following methods to handle execution errors:

- Add a `Catch` block in the state definition.
- Check the value of the `hasExecutionErrors` value in the state machine context.

Catch

To use `Catch`, add the following to your state definition:

```
"Catch": [
    {
```

```
        "ErrorEquals": [
            "<error-type>"
        ]
        "Next": "<state-name>"
    }
]
```

All fields that contain values are required as described here:

`Catch.ErrorEquals`

An array of the error types to catch. If an execution error matches one of the specified values, then the state machine transitions to the state specified in `Catch.Next`. See each state definition for information about the type of error it produces.

`Catch.Next`

The next state to transition to if the current state encounters an execution error that matches one of the values specified in `Catch.ErrorEquals` .

Catch blocks are handled sequentially until one matches. If the no errors match the ones listed in the Catch blocks, then the state machines continues to execute. Because execution errors are a result of incorrect state definitions, we recommend that you transition to the Fail state when a state encounters an execution error.

hasExecutionError

When some states encounter execution errors, in addition to issuing the error, they also set the `hasExecutionError` value to `true` in the state machine context. You can use this value to detect when an error occurs, and then use a `Choice` state to transition the state machine to the `Fail` state.

This method has the following characteristics.

- The state machine does not start with any value assigned to `hasExecutionError`, and this value is not available until a particular state sets it. This means that you must explicitly set the `FallthroughOnError` to `false` for the `Choice` states that access this value to prevent the state machine from stopping if no execution errors occur.
- Once it is set to `true`, `hasExecutionError` is never set to false or removed from the context. This means that this value is useful only the first time that it is set to `true`, and for all subsequent states, it does not provide a meaningful value.

- The `hasExecutionError` value is shared with all branch state machines in the `Parallel` state, which can result in unexpected results depending on the order in which it is accessed.

Because of these characteristics, we do not recommend that you use this method if you can use a Catch block instead.

Example state machines

This section provides some example state machine configurations.

Examples

- Example state machine: Run a single test group
- Example state machine: Run user-selected test groups
- Example state machine: Run a single test group with product features
- Example state machine: Run two test groups in parallel

Example state machine: Run a single test group

This state machine:

- Runs the test group with id `GroupA`, which must be present in the suite in a `group.json` file.
- Checks for execution errors and transitions to `Fail` if any are found.
- Generates a report and transitions to `Succeed` if there are no errors, and `Fail` otherwise.

```
{
    "Comment": "Runs a single group and then generates a report.",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
```

```
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Example state machine: Run user-selected test groups

This state machine:

- Checks if the test runner selected specific test groups. The state machine does not check for specific test cases because test runners cannot select test cases without also selecting a test group.
- If test groups are selected:
  - Runs the test cases within the selected test groups. To do so, the state machine does not explicitly specify any test groups or test cases in the `RunTask` state.
  - Generates a report after running all tests and exits.
- If test groups are not selected:
  - Runs tests in test group `GroupA`.
  - Generates reports and exits.

```
{
    "Comment": "Runs specific groups if the test runner chose to do that, otherwise
 runs GroupA.",
    "StartAt": "SpecificGroupsCheck",
    "States": {
        "SpecificGroupsCheck": {
            "Type": "Choice",
            "Default": "RunGroupA",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.specificTestGroups[0]}} != ''",
                    "Next": "RunSpecificGroups"
                }
            ]
        },
        "RunSpecificGroups": {
            "Type": "RunTask",
            "Next": "Report",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Report": {
            "Type": "Report",
```

```
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Example state machine: Run a single test group with product features

This state machine:

- Runs the test group `GroupA`.
- Checks for execution errors and transitions to `Fail` if any are found.
- Adds the `FeatureThatDependsOnGroupA` feature to the `awsiotdevicetester_report.xml` file:
  - If `GroupA` passes, the feature is set to `supported`.
  - The feature is not marked optional in the report.
- Generates a report and transitions to `Succeed` if there are no errors, and `Fail` otherwise

```
{
    "Comment": "Runs GroupA and adds product features based on GroupA",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "AddProductFeatures",
            "TestGroup": "GroupA",
            "ResultVar": "GroupA_passed",
```

```
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Example state machine: Run two test groups in parallel

This state machine:

- Runs the `GroupA` and `GroupB` test groups in parallel. The `ResultVar` variables stored in the context by the `RunTask` states in the branch state machines by are available to the `AddProductFeatures` state.
- Checks for execution errors and transitions to `Fail` if any are found. This state machine does not use a `Catch` block because that method does not detect execution errors in branch state machines.
- Adds features to the `awsiotdevicetester_report.xml` file based on the groups that pass
  - If `GroupA` passes, the feature is set to `supported`.
  - The feature is not marked optional in the report.
- Generates a report and transitions to `Succeed` if there are no errors, and `Fail` otherwise

If two devices are configured in the device pool, both `GroupA` and `GroupB` can run at the same time. However, if either `GroupA` or `GroupB` has multiple tests in it, then both devices may be allocated to those tests. If only one device is configured, the test groups will run sequentially.

```
{
    "Comment": "Runs GroupA and GroupB in parallel",
    "StartAt": "RunGroupAAndB",
    "States": {
        "RunGroupAAndB": {
            "Type": "Parallel",
            "Next": "CheckForErrors",
            "Branches": [
                {
                    "Comment": "Run GroupA state machine",
                    "StartAt": "RunGroupA",
                    "States": {
                        "RunGroupA": {
                            "Type": "RunTask",
                            "Next": "Succeed",
                            "TestGroup": "GroupA",
                            "ResultVar": "GroupA_passed",
                            "Catch": [
                                {
                                    "ErrorEquals": [
```

```
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            },
            {
                "Comment": "Run GroupB state machine",
                "StartAt": "RunGroupB",
                "States": {
                    "RunGroupA": {
                        "Type": "RunTask",
                        "Next": "Succeed",
                        "TestGroup": "GroupB",
                        "ResultVar": "GroupB_passed",
                        "Catch": [
                            {
                                "ErrorEquals": [
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            }
        ]
    },
    "CheckForErrors": {
        "Type": "Choice",
```

```
            "Default": "AddProductFeatures",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.hasExecutionErrors}} == true",
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                },
                {
                    "Feature": "FeatureThatDependsOnGroupB",
                    "Groups": [
                        "GroupB"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
```

```
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Create IDT test case executable

You can create and place test case executable in a test suite folder in the following ways:

- For test suites that use arguments or environment variables from the `test.json` files to determine which tests to run, you can create a single test case executable for the entire test suite, or a test executable for each test group in the test suite.
- For a test suite where you want to run specific tests based on specified commands, you create one test case executable for each test case in the test suite.

As a test writer, you can determine which approach is appropriate for your use case and structure your test case executable accordingly. Make sure that your provide the correct test case executable path in each `test.json` file, and that the specified executable runs correctly.

When all devices are ready for a test case to run, IDT reads the following files:

- The `test.json` for the selected test case determines the processes to start and the environment variables to set.
- The `suite.json` for the test suite determines the environment variables to set.

IDT starts the required test executable process based on the commands and arguments specified in the `test.json` file, and passes the required environment variables to the process.

Use the IDT Client SDK

The IDT Client SDKs let you simplify how you write test logic in your test executable with API commands that you can use interact with IDT and your devices under test. IDT currently provides the following SDKs:

- IDT Client SDK for Python
- IDT Client SDK for Go
- IDT Client SDK for Java

These SDKs are located in the *<device-tester-extract-location>*/sdks folder. When you create a new test case executable, you must copy the SDK that you want to use to the folder that contains your test case executable and reference the SDK in your code. This section provides a brief description of the available API commands that you can use in your test case executables.

In this section

- Device interaction
- IDT interaction
- Host interaction

Device interaction

The following commands enable you to communicate with the device under test without having to implement any additional device interaction and connectivity management functions.

`ExecuteOnDevice`

Allows test suites to run shell commands on a device that support SSH or Docker shell connections.

`CopyToDevice`

Allows test suites to copy a local file from the host machine that runs IDT to a specified location on a device that supports SSH or Docker shell connections.

`ReadFromDevice`

Allows test suites to read from the serial port of devices that support UART connections.

> **Note**
>
> Because IDT does not manage direct connections to devices that are made using device access information from the context, we recommend using these device interaction API commands in your test case executables. However, if these commands do not meet your test case requirements, then you can retrieve device access information from the IDT context and use it to make a direct connection to the device from the test suite.
> To make a direct connection, retrieve the information in the `device.connectivity` and the `resource.devices.connectivity` fields for your device under test and for resource

devices, respectively. For more information about using the IDT context, see Use the IDT context.

IDT interaction

The following commands enable your test suites to communicate with IDT.

`PollForNotifications`

Allows test suites to check for notifications from IDT.

`GetContextValue` and `GetContextString`

Allows test suites to retrieve values from the IDT context. For more information, see Use the IDT context.

`SendResult`

Allows test suites to report test case results to IDT. This command must be called at the end of each test case in a test suite.

Host interaction

The following command enable your test suites to communicate with the host machine.

`PollForNotifications`

Allows test suites to check for notifications from IDT.

`GetContextValue` and `GetContextString`

Allows test suites to retrieve values from the IDT context. For more information, see Use the IDT context.

`ExecuteOnHost`

Allows test suites to run commands on the local machine and lets IDT manage the test case executable lifecycle.

Enable IDT CLI commands

The `run-suite` command IDT CLI provides several options that let test runner customize test execution. To allow test runners to use these options to run your custom test suite, you implement

support for the IDT CLI. If you do not implement support, test runners will still be able to run tests, but some CLI options will not function correctly. To provide an ideal customer experience, we recommend that you implement support for the following arguments for the `run-suite` command in the IDT CLI:

`timeout-multiplier`

Specifies a value greater than 1.0 that will be applied to all timeouts while running tests.

Test runners can use this argument to increase the timeout for the test cases that they want to run. When a test runner specifies this argument in their `run-suite` command, IDT uses it to calculate the value of the IDT_TEST_TIMEOUT environment variable and sets the `config.timeoutMultiplier` field in the IDT context. To support this argument, you must do the following:

- Instead of directly using the timeout value from the `test.json` file, read the IDT_TEST_TIMEOUT environment variable to obtain the correctly calculated timeout value.
- Retrieve the `config.timeoutMultiplier` value from the IDT context and apply it to long running timeouts.

For more information about exiting early because of timeout events, see Specify exit behavior.

`stop-on-first-failure`

Specifies that IDT should stop running all tests if it encounters a failure.

When a test runner specifies this argument in their `run-suite` command, IDT will stop running tests as soon as it encounters a failure. However, if test cases are running in parallel, then this can lead to unexpected results. To implement support, make sure that if IDT encounters this event, your test logic instructs all running test cases to stop, clean up temporary resources, and report a test result to IDT. For more information about exiting early on failures, see Specify exit behavior.

`group-id` and `test-id`

Specifies that IDT should run only the selected test groups or test cases.

Test runners can use these arguments with their `run-suite` command to specify the following test execution behavior:

- Run all tests inside the specified test groups.
- Run a selection of tests from within a specified test group.

To support these arguments, the state machine for your test suite must include a specific set of `RunTask` and `Choice` states in your state machine. If you are not using a custom state machine, then the default IDT state machine includes the required states for you and you do not need to take additional action. However, if you are using a custom state machine, then use Example state machine: Run user-selected test groups as a sample to add the required states in your state machine.

For more information about IDT CLI commands, see Debug and run custom test suites.

Write event logs

While the test is running, you send data to `stdout` and `stderr` to write event logs and error messages to the console. For information about the format of console messages, see Console message format.

When the IDT finishes running the test suite, this information is also available in the `test_manager.log` file located in the *<devicetester-extract-location>*/`results/`*<execution-id>*`/logs` folder.

You can configure each test case to write the logs from its test run, including logs from the device under test, to the *<group-id>*`_`*<test-id>* file located in the *<device-tester-extract-location>*`/results/`*execution-id*`/logs` folder. To do this, retrieve the path to the log file from the IDT context with the `testData.logFilePath` query, create a file at that path, and write the content that you want to it. IDT automatically updates the path based on the test case that is running. If you choose not to create the log file for a test case, then no file is generated for that test case.

You can also set up your text executable to create additional log files as needed in the *<device-tester-extract-location>*`/logs` folder. We recommend that you specify unique prefixes for log file names so your files don't get overwritten.

Report results to IDT

IDT writes test results to the `awsiotdevicetester_report.xml` and the *suite-name*`_report.xml` files. These report files are located in *<device-tester-extract-location>*`/results/`*<execution-id>*`/`. Both reports capture the results from the test suite execution. For more information about the schemas that IDT uses for these reports, see Review IDT test results and logs

To populate the contents of the *suite-name*_report.xml file, you must use the SendResult command to report test results to IDT before the test execution finishes. If IDT cannot locate the results of a test, it issues an error for the test case. The following Python excerpt shows the commands to send a test result to IDT:

```
request-variable = SendResultRequest(TestResult(result))
client.send_result(request-variable)
```

If you do not report results through the API, IDT looks for test results in the test artifacts folder. The path to this folder is stored in the testData.testArtifactsPath filed in the IDT context. In this folder, IDT uses the first alphabetically sorted XML file it locates as the test result.

If your test logic produces JUnit XML results, you can write the test results to an XML file in the artifacts folder to directly provide the results to IDT instead of parsing the results and then using the API to submit them to IDT.

If you use this method, make sure that your test logic accurately summarizes the test results and format your result file in the same format as the *suite-name*_report.xml file. IDT does not perform any validation of the data that you provide, with the following exceptions:

- IDT ignores all properties of the testsuites tag. Instead, it calculates the tag properties from other reported test group results.
- At least one testsuite tag must exist within testsuites.

Because IDT uses the same artifacts folder for all test cases and does not delete result files between test runs, this method might also lead to erroneous reporting if IDT reads the incorrect file. We recommend that you use the same name for the generated XML results file across all test cases to overwrite the results for each test case and make sure that the correct results are available for IDT to use. Although you can use a mixed approach to reporting in your test suite, that is, use an XML result file for some test cases and submit results through the API for others, we do not recommend this approach.

Specify exit behavior

Configure your text executable to always exit with an exit code of 0, even if a test case reports a failure or an error result. Use non-zero exit codes only to indicate that a test case did not run or if the test case executable could not communicate any results to IDT. When IDT receives a non-zero exit code, it marks the test case has having encountered an error that prevented it from running.

IDT might request or expect a test case to stop running before it has finished in the following events. Use this information to configure your test case executable to detect each of these events from the test case:

Timeout

Occurs when a test case runs for longer than the timeout value specified in the `test.json` file. If the test runner used the `timeout-multiplier` argument to specify a timeout multiplier, then IDT calculates the timeout value with the multiplier.

To detect this event, use the IDT_TEST_TIMEOUT environment variable. When a test runner launches a test, IDT sets the value of the IDT_TEST_TIMEOUT environment variable to the calculated timeout value (in seconds) and passes the variable to the test case executable. You can read the variable value to set an appropriate timer.

Interrupt

Occurs when the test runner interrupts IDT. For example, by pressing **Ctrl+C**.

Because terminals propagate signals to all child processes, you can simply configure a signal handler in your test cases to detect interrupt signals.

Alternatively, you can periodically poll the API to check the value of the `CancellationRequested` boolean in the `PollForNotifications` API response. When IDT receives an interrupt signal, it sets the value of the `CancellationRequested` boolean to `true`.

Stop on first failure

Occurs when a test case that is running in parallel with the current test case fails and the test runner used the `stop-on-first-failure` argument to specify that IDT should stop when it encounters any failure.

To detect this event, you can periodically poll the API to check the value of the `CancellationRequested` boolean in the `PollForNotifications` API response. When IDT encounters a failure and is configured to stop on first failure, it sets the value of the `CancellationRequested` boolean to `true`.

When any of these events occur, IDT waits for 5 minutes for any currently running test cases to finish running. If all running test cases do not exit within 5 minutes, IDT forces each of their processes to stop. If IDT has not received test results before the processes end, it will mark the test

cases as having timed out. As a best practice, you should ensure that your test cases perform the following actions when they encounter one of the events:

1. Stop running normal test logic.
2. Clean up any temporary resources, such as test artifacts on the device under test.
3. Report a test result to IDT, such as a test failure or an error.
4. Exit.

Use the IDT context

When IDT runs a test suite, the test suite can access a set of data that can be used to determine how each test runs. This data is called the IDT context. For example, user data configuration provided by test runners in a `userdata.json` file is made available to test suites in the IDT context.

The IDT context can be considered a read-only JSON document. Test suites can retrieve data from and write data to the context using standard JSON data types like objects, arrays, numbers and so on.

Context schema

The IDT context uses the following format:

```
{
    "config": {
        <config-json-content>
        "timeoutMultiplier": timeout-multiplier,
        "idtRootPath": <path/to/IDT/root>
    },
    "device": {
        <device-json-device-element>
    },
    "devicePool": {
        <device-json-pool-element>
    },
    "resource": {
        "devices": [
            {
                <resource-json-device-element>
                "name": "<resource-name>"
```

```
            }
        ]
    },
    "testData": {
        "awsCredentials": {
            "awsAccessKeyId": "<access-key-id>",
            "awsSecretAccessKey": "<secret-access-key>",
            "awsSessionToken": "<session-token>"
        },
        "logFilePath": "/path/to/log/file"
    },
    "userData": {
        <userdata-json-content>
    }
}
```

`config`

Information from the `config.json` file. The `config` field also contains the following additional fields:

`config.timeoutMultiplier`

The multiplier for the any timeout value used by the test suite. This value is specified by the test runner from the IDT CLI. The default value is 1.

`config.idRootPath`

This value is a placeholder for the absolute path value of IDT while configuring the `userdata.json` file. This is used by the build and flash commands.

`device`

Information about the device selected for the test run. This information is equivalent to the `devices` array element in the `device.json` file for the selected device.

`devicePool`

Information about the device pool selected for the test run. This information is equivalent to the top-level device pool array element defined in the `device.json` file for the selected device pool.

`resource`

Information about resource devices from the `resource.json` file.

`resource.devices`

This information is equivalent to the `devices` array defined in the `resource.json` file. Each `devices` element includes the following additional field:

`resource.device.name`

The name of the resource device. This value is set to the `requiredResource.name` value in the `test.json` file.

`testData.awsCredentials`

The AWS credentials used by the test to connect to the AWS cloud. This information is obtained from the `config.json` file.

`testData.logFilePath`

The path to the log file to which the test case writes log messages. The test suite creates this file if it doesn't exist.

`userData`

Information provided by the test runner in the `userdata.json` file.

Access data in the context

You can query the context using JSONPath notation from your configuration files and from your text executable with the `GetContextValue` and `GetContextString` APIs. The syntax for JSONPath strings to access the IDT context varies as follows:

- In `suite.json` and `test.json`, you use {{*query*}}. That is, do not use the root element `$.` to start your expression.
- In `statemachine.json`, you use {{`$.`*query*}}.
- In API commands, you use *query* or {{`$.`*query*}}, depending on the command. For more information, see the inline documentation in the SDKs.

The following table describes the operators in a typical foobar JSONPath expression:

Operator	Description
`$`	The root element. Because the top-level context value for IDT is an object, you will typically use `$.` to start your queries.
`.childName`	Accesses the child element with name `childName` from an object. If applied to an array, yields a new array with this operator applied to each element. The element name is case sensitive. For example, the query to access the `awsRegion` value in the `config` object is `$.config.awsRegion` .
`[start:end]`	Filters elements from an array, retrieving items beginning from the `start` index and going up to the `end` index, both inclusive.
`[index1, index2, ... , indexN]`	Filters elements from an array, retrieving items from only the specified indices.
`[?(expr)]`	Filters elements from an array using the `expr` expression. This expression must evaluate to a boolean value.

To create filter expressions, use the following syntax:

```
<jsonpath> | <value> operator <jsonpath> | <value>
```

In this syntax:

- `jsonpath` is a JSONPath that uses standard JSON syntax.
- `value` is any custom value that uses standard JSON syntax.
- `operator` is one of the following operators:
  - < (Less than)
  - <= (Less than or equal to)

- == (Equal to)

  If the JSONPath or value in your expression is an array, boolean, or object value, then this is the only supported binary operator that you can use.
- >= (Greater than or equal to)
- > (Greater than)
- =~ (Regular expression match). To use this operator in a filter expression, the JSONPath or value on the left side of your expression must evaluate to a string and the right side must be a pattern value that follows the RE2 syntax.

You can use JSONPath queries in the form {{*query*}} as placeholder strings within the `args` and `environmentVariables` fields in `test.json` files and within the `environmentVariables` fields in `suite.json` files. IDT performs a context lookup and populates the fields with the evaluated value of the query. For example, in the `suite.json` file, you can use placeholder strings to specify environment variable values that change with each test case and IDT will populate the environment variables with the correct value for each test case. However, when you use placeholder strings in `test.json` and `suite.json` files, the following considerations apply for your queries:

- You must each occurrence of the `devicePool` key in your query in all lower case. That is, use `devicepool` instead.
- For arrays, you can use only arrays of strings. In addition, arrays use a non-standard `item1,item2,...,itemN` format. If the array contains only one element, then it is serialized as `item`, making it indistinguishable from a string field.
- You cannot use placeholders to retrieve objects from the context.

Because of these considerations, we recommend that whenever possible, you use the API to access the context in your test logic instead of placeholder strings in `test.json` and `suite.json` files. However, in some cases it might be more convenient to use JSONPath placeholders to retrieve single strings to set as environment variables.

Configure settings for test runners

To run custom test suites, test runners must configure their settings based on the test suite that they want to run. Settings are specified based on configuration file templates located in the

<device-tester-extract-location>/configs/ folder. If required, test runners must also set up AWS credentials that IDT will use to connect to the AWS cloud.

As a test writer, you will need to configure these files to debug your test suite. You must provide instructions to test runners so that they can configure the following settings as needed to run your test suites.

Configure device.json

The `device.json` file contains information about the devices that tests are run on (for example, IP address, login information, operating system, and CPU architecture).

Test runners can provide this information using the following template `device.json` file located in the *<device-tester-extract-location>*/configs/ folder.

```
[
    {
        "id": "<pool-id>",
        "sku": "<pool-sku>",
        "features": [
            {
                "name": "<feature-name>",
                "value": "<feature-value>",
                "configs": [
                    {
                        "name": "<config-name>",
                        "value": "<config-value>"
                    }
                ],
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "pairedResource": "<device-id>", //used for no-op protocol
                "connectivity": {
                    "protocol": "ssh | uart | docker | no-op",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "publicKeyPath": "<public-key-path>",
                    "auth": {
                        "method": "pki | password",
```

```
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

All fields that contain values are required as described here:

`id`

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

`sku`

An alphanumeric value that uniquely identifies the device under test. The SKU is used to track qualified devices.

> **Note**
>
> If you want to list your board in the AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

`features`

Optional. An array that contains the device's supported features. Device features are user-defined values that you configure in your test suite. You must provide your test runners with information about the feature names and values to include in the `device.json` file. For example, if you want to test a device that functions as an MQTT server for other devices, then you can configure your test logic to validate specific supported levels for a feature named `MQTT_QoS`. Test runners provide this feature name and set the feature value to the QoS levels supported by their device. You can retrieve the provided information from the IDT context with the `devicePool.features` query, or from the state machine context with the `pool.features` query.

`features.name`

The name of the feature.

`features.value`

The supported feature values.

`features.configs`

Configuration settings, if needed, for the feature.

`features.config.name`

The name of the configuration setting.

`features.config.value`

The supported setting values.

`devices`

An array of devices in the pool to be tested. At least one device is required.

`devices.id`

A user-defined unique identifier for the device being tested.

`devices.pairedResource`

A user-defined unique identifier for a resource device. This value is required when you test devices using the `no-op` connectivity protocol.

`connectivity.protocol`

The communication protocol used to communicate with this device. Each device in a pool must use the same protocol.

Currently, the only supported values are `ssh` and `uart` for physical devices, `docker` for Docker containers, and `no-op` for devices who don't have a direct connection with the IDT host machine but require a resource device as physical middleware to communicate with the host machine.

For no-op devices, you configure the resource device ID in `devices.pairedResource`. You must also specify this ID in the `resource.json` file. The paired device must be a device that is physically paired with the device under test. After IDT identifies and connects to the paired resource device, IDT will not connect to other resource devices according to the features described in the `test.json` file.

`connectivity.ip`

The IP address of the device being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.port`

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.publicKeyPath`

Optional. The full path to the public key used to authenticate connections to the device under test. When you specify the `publicKeyPath`, IDT validates the device's public key when it establishes an SSH connection to the device under test. If this value is not specified, IDT creates an SSH connection, but doesn't validate the device's public key.

We strongly recommend that you specify the path to the public key, and that you use a secure method to fetch this public key. For standard command line-based SSH clients, the public key is provided in the `known_hosts` file. If you specify a separate public key file, this file must use the same format as the `known_hosts` file, that is, `ip-address key-type public-key`.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

`connectivity.serialPort`

Optional. The serial port to which the device is connected.

This property applies only if `connectivity.protocol` is set to `uart`.

`connectivity.containerId`

The container ID or name of the Docker container being tested.

This property applies only if `connectivity.protocol` is set to `docker`.

`connectivity.containerUser`

Optional. The name of the user to user inside the container. The default value is the user provided in the Dockerfile.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `docker`.

> **Note**
>
> To check if test runners configure the incorrect device connection for a test, you can retrieve `pool.Devices[0].Connectivity.Protocol` from the state machine context and compare it to the expected value in a `Choice` state. If an incorrect protocol is used, then print a message using the `LogMessage` state and transition to the `Fail` state.
> Alternatively, you can use error handling code to report a test failure for incorrect device types.

(Optional) Configure userdata.json

The `userdata.json` file contains any additional information that is required by a test suite but is not specified in the `device.json` file. The format of this file depends on the `userdata_scheme.json` file that is defined in the test suite. If you are a test writer, make sure you provide this information to users who will run the test suites that you write.

(Optional) Configure resource.json

The `resource.json` file contains information about any devices that will be used as resource devices. Resource devices are devices that are required to test certain capabilities of a device under test. For example, to test a device's Bluetooth capability, you might use a resource device to test that your device can connect to it successfully. Resource devices are optional, and you can require as many resources devices as you need. As a test writer, you use the test.json file to define the resource device features that are required for a test. Test runners then use the `resource.json` file to provide a pool of resource devices that have the required features. Make sure you provide this information to users who will run the test suites that you write.

Test runners can provide this information using the following template `resource.json` file located in the *`<device-tester-extract-location>`*`/configs/` folder.

```
[
    {
        "id": "<pool-id>",
        "features": [
            {
                "name": "<feature-name>",
                "version": "<feature-value>",
                "jobSlots": <job-slots>
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "publicKeyPath": "<public-key-path>",
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

All fields that contain values are required as described here:

`id`

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

`features`

Optional. An array that contains the device's supported features. The information required in this field is defined in the test.json files in the test suite and determines which tests to run and how to run those tests. If the test suite does not require any features, then this field is not required.

`features.name`

The name of the feature.

`features.version`

The feature version.

`features.jobSlots`

Setting to indicate how many tests can concurrently use the device. The default value is 1.

`devices`

An array of devices in the pool to be tested. At least one device is required.

`devices.id`

A user-defined unique identifier for the device being tested.

`connectivity.protocol`

The communication protocol used to communicate with this device. Each device in a pool must use the same protocol.

Currently, the only supported values are `ssh` and `uart` for physical devices, and `docker` for Docker containers.

`connectivity.ip`

The IP address of the device being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.port`

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.publicKeyPath`

Optional. The full path to the public key used to authenticate connections to the device under test. When you specify the `publicKeyPath`, IDT validates the device's public key when it establishes an SSH connection to the device under test. If this value is not specified, IDT creates an SSH connection, but doesn't validate the device's public key.

We strongly recommend that you specify the path to the public key, and that you use a secure method to fetch this public key. For standard command line-based SSH clients, the public key is provided in the `known_hosts` file. If you specify a separate public key file, this file must use the same format as the `known_hosts` file, that is, `ip-address key-type public-key`.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

`connectivity.serialPort`

Optional. The serial port to which the device is connected.

This property applies only if `connectivity.protocol` is set to `uart`.

`connectivity.containerId`

The container ID or name of the Docker container being tested.

This property applies only if `connectivity.protocol` is set to `docker`.

`connectivity.containerUser`

Optional. The name of the user to user inside the container. The default value is the user provided in the Dockerfile.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `docker`.

(Optional) Configure config.json

The `config.json` file contains configuration information for IDT. Typically, test runners will not need to modify this file except to provide their AWS user credentials for IDT, and optionally, an AWS region. If AWS credentials with required permissions are provided AWS IoT Device Tester collects and submits usage metrics to AWS. This is an opt-in feature and is used to improve IDT functionality. For more information, see Submit IDT usage metrics.

Test runners can configure their AWS credentials in one of the following ways:

- **Credentials file**

IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

The location of the credentials file varies, depending on the operating system you are using:

- macOS, Linux: `~/.aws/credentials`
- Windows: `C:\Users\UserName\.aws\credentials`

- **Environment variables**

  Environment variables are variables maintained by the operating system and used by system commands. Variables defined during an SSH session are not available after that session is closed. IDT can use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to store AWS credentials

  To set these variables on Linux, macOS, or Unix, use **export**:

  ```
  export AWS_ACCESS_KEY_ID=<your_access_key_id>
  export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
  ```

  To set these variables on Windows, use **set**:

  ```
  set AWS_ACCESS_KEY_ID=<your_access_key_id>
  set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
  ```

To configure AWS credentials for IDT, test runners edit the `auth` section in the `config.json` file located in the *<device-tester-extract-location>*`/configs/` folder.

```
{
    "log": {
        "location": "logs"
    },
    "configFiles": {
        "root": "configs",
        "device": "configs/device.json"
    },
    "testPath": "tests",
    "reportPath": "results",
    "awsRegion": "<region>",
    "auth": {
```

```
        "method": "file | environment",
        "credentials": {
            "profile": "<profile-name>"
        }
    }
}
]
```

All fields that contain values are required as described here:

> **Note**
>
> All paths in this file are defined relative to the *<device-tester-extract-location>*.

`log.location`

The path to the logs folder in the *<device-tester-extract-location>*.

`configFiles.root`

The path to the folder that contains the configuration files.

`configFiles.device`

The path to the `device.json` file.

`testPath`

The path to the folder that contains test suites.

`reportPath`

The path to the folder that will contain test results after IDT runs a test suite.

`awsRegion`

Optional. The AWS region that test suites will use. If not set, then test suites will use the default region specified in each test suite.

`auth.method`

The method IDT uses to retrieve AWS credentials. Supported values are `file` to retrieve credentials from a credentials file, and `environment` to retrieve credentials using environment variables.

`auth.credentials.profile`

The credentials profile to use from the credentials file. This property applies only if `auth.method` is set to `file`.

Debug and run custom test suites

After the required configuration is set, IDT can run your test suite. The runtime of the full test suite depends on the hardware and the composition of the test suite. For reference, it takes approximately 30 minutes to complete the full FreeRTOS qualification test suite on a Raspberry Pi 3B.

As you write your test suite, you can use IDT to run the test suite in debug mode to check your code before you run it or provide it to test runners.

Run IDT in debug mode

Because test suites depend on IDT to interact with devices, provide the context, and receive results, you cannot simply debug your test suites in an IDE without any IDT interaction. To do so, the IDT CLI provides the `debug-test-suite` command that lets you run IDT in debug mode. Run the following command to view the available options for `debug-test-suite`:

```
devicetester_[linux | mac | win_x86-64] debug-test-suite -h
```

When you run IDT in debug mode, IDT does not actually launch the test suite or run the test orchestrator; instead, it interacts with your IDE to responds to requests made from the test suite running in the IDE and prints the logs to the console. IDT does not time out and waits to exit until manually interrupted. In debug mode, IDT also does not run the test orchestrator and will not generate any report files. To debug your test suite, you must use your IDE to provide some information that IDT usually obtains from the configuration files. Make sure you provide the following information:

- Environment variables and arguments for each test. IDT will not read this information from `test.json` or `suite.json`.
- Arguments to select resource devices. IDT will not read this information from `test.json`.

To debug your test suites, complete the following steps:

1. Create the setting configuration files that are required to run the test suite. For example, if your test suite requires the `device.json`, `resource.json`, and `user data.json`, make sure you configure all of them as needed.
2. Run the following command to place IDT in debug mode and select any devices that are required to run the test.

   ```
   devicetester_[linux | mac | win_x86-64] debug-test-suite [options]
   ```

   After you run this command, IDT waits for requests from the test suite and then responds to them. IDT also generates the environment variables that are required for the case process for the IDT Client SDK.
3. In your IDE, use the `run` or `debug` configuration to do the following:

   a. Set the values of the IDT-generated environment variables.

   b. Set the value of any environment variables or arguments that you specified in your `test.json` and `suite.json` file.

   c. Set breakpoints as needed.
4. Run the test suite in your IDE.

   You can debug and re-run the test suite as many times as needed. IDT does not time out in debug mode.
5. After you complete debugging, interrupt IDT to exit debug mode.

IDT CLI commands to run tests

The following section describes the IDT CLI commands:

IDT v4.0.0

`help`

Lists information about the specified command.

`list-groups`

Lists the groups in a given test suite.

`list-suites`

Lists the available test suites.

`list-supported-products`

Lists the supported products for your version of IDT, in this case FreeRTOS versions, and FreeRTOS qualification test suite versions available for the current IDT version.

`list-test-cases`

Lists the test cases in a given test group. The following option is supported:

- `group-id`. The test group to search for. This option is required and must specify a single group.

`run-suite`

Runs a suite of tests on a pool of devices. The following are some commonly used options:

- `suite-id`. The test suite version to run. If not specified, IDT uses the latest version in the `tests` folder.
- `group-id`. The test groups to run, as a comma-separated list. If not specified, IDT runs all test groups in the test suite.
- `test-id`. The test cases to run, as a comma-separated list. When specified, `group-id` must specify a single group.
- `pool-id`. The device pool to test. Test runners must specify a pool if they have multiple device pools defined in your `device.json` file.
- `timeout-multiplier`. Configures IDT to modify the test execution timeout specified in the `test.json` file for a test with a user-defined multiplier.
- `stop-on-first-failure`. Configures IDT to stop execution on the first failure. This option should be used with `group-id` to debug the specified test groups.
- `userdata`. Sets the file that contains user data information required to run the test suite. This is required only if `userdataRequired` is set to true in the `suite.json` file for the test suite.

For more information about `run-suite` options, use the `help` option:

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

`debug-test-suite`

Run the test suite in debug mode. For more information, see Run IDT in debug mode.

Review IDT test results and logs

This section describes the format in which IDT generates console logs and test reports.

Console message format

AWS IoT Device Tester uses a standard format for printing messages to the console when it starts a test suite. The following excerpt shows an example of a console message generated by IDT.

```
[INFO] [2000-01-02 03:04:05]: Using suite: MyTestSuite_1.0.0
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

Most console messages consist of the following fields:

`time`

A full ISO 8601 timestamp for the logged event.

`level`

The message level for the logged event. Typically, the logged message level is one of `info`, `warn`, or `error`. IDT issues a `fatal` or `panic` message if it encounters an expected event that causes it to exit early.

`msg`

The logged message.

`executionId`

A unique ID string for the current IDT process. This ID is used to differentiate between individual IDT runs.

Console messages generated from a test suite provide additional information about the device under test and the test suite, test group, and test cases that IDT runs. The following excerpt shows an example of a console message generated from a test suite.

```
[INFO] [2000-01-02 03:04:05]: Hello world! suiteId=MyTestSuitegroupId=myTestGroup
 testCaseId=myTestCase deviceId=my-
deviceexecutionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

The test-suite specific part of the console message contains the following fields:

`suiteId`

The name of the test suite currently running.

`groupId`

The ID of the test group currently running.

`testCaseId`

The ID of the test case current running.

`deviceId`

A ID of the device under test that the current test case is using.

The test summary contains information about the test suite, the test results for each group that was run, and the locations of the generated logs and report files. The following example shows a test summary message.

```
========== Test Summary ==========
Execution Time:     5m00s
Tests Completed:    4
Tests Passed:       3
Tests Failed:       1
Tests Skipped:      0
----------------------------------
Test Groups:
    GroupA:         PASSED
    GroupB:         FAILED
----------------------------------
Failed Tests:
    Group Name: GroupB
        Test Name: TestB1
            Reason: Something bad happened
----------------------------------
Path to AWS IoT Device Tester Report: /path/to/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/logs
Path to Aggregated JUnit Report: /path/to/MyTestSuite_Report.xml
```

AWS IoT Device Tester report schema

`awsiotdevicetester_report.xml` is a signed report that contains the following information:

- The IDT version.
- The test suite version.
- The report signature and key used to sign the report.
- The device SKU and the device pool name specified in the `device.json` file.
- The product version and the device features that were tested.
- The aggregate summary of test results. This information is the same as that contained in the *suite-name*`_report.xml` file.

```
<apnreport>
    <awsiotdevicetesterversion>idt-version</awsiotdevicetesterversion>
    <testsuiteversion>test-suite-version</testsuiteversion>
    <signature>signature</signature>
    <keyname>keyname</keyname>
    <session>
        <testsession>execution-id</testsession>
        <starttime>start-time</starttime>
        <endtime>end-time</endtime>
    </session>
    <awsproduct>
        <name>product-name</name>
        <version>product-version</version>
        <features>
            <feature name="<feature-name>" value="supported | not-supported | <feature-
value>" type="optional | required"/>
        </features>
    </awsproduct>
    <device>
        <sku>device-sku</sku>
        <name>device-name</name>
        <features>
            <feature name="<feature-name>" value="<feature-value>"/>
        </features>
        <executionMethod>ssh | uart | docker</executionMethod>
    </device>
    <devenvironment>
        <os name="<os-name>"/>
    </devenvironment>
    <report>
        <suite-name-report-contents>
    </report>
```

```
</apnreport>
```

The `awsiotdevicetester_report.xml` file contains an `<awsproduct>` tag that contains information about the product being tested and the product features that were validated after running a suite of tests.

Attributes used in the `<awsproduct>` tag

`name`

The name of the product being tested.

`version`

The version of the product being tested.

`features`

The features validated. Features marked as `required` are required for the test suite to validate the device. The following snippet shows how this information appears in the `awsiotdevicetester_report.xml` file.

```
<feature name="ssh" value="supported" type="required"></feature>
```

Features marked as `optional` are not required for validation. The following snippets show optional features.

```
<feature name="hsi" value="supported" type="optional"></feature>
<feature name="mqtt" value="not-supported" type="optional"></feature>
```

Test suite report schema

The *`suite-name`*`_Result.xml` report is in JUnit XML format. You can integrate it into continuous integration and deployment platforms like Jenkins, Bamboo, and so on. The report contains an aggregate summary of test results.

```
<testsuites name="<suite-name> results" time="<run-duration>" tests="<number-of-test>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
    <testsuite name="<test-group-id>" package="" tests="<number-of-tests>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
```

```
        <!--success-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>"/>
        <!--failure-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <failure type="<failure-type>">
                reason
            </failure>
        </testcase>
        <!--skipped-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <skipped>
                reason
            </skipped>
        </testcase>
        <!--error-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <error>
                reason
            </error>
        </testcase>
    </testsuite>
</testsuites>
```

The report section in both the `awsiotdevicetester_report.xml` or *`suite-name`*`_report.xml` lists the tests that were run and the results.

The first XML tag `<testsuites>` contains the summary of the test execution. For example:

```
<testsuites name="MyTestSuite results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

Attributes used in the `<testsuites>` tag

`name`

The name of the test suite.

`time`

The time, in seconds, it took to run the test suite.

`tests`

The number of tests executed.

`failures`

The number of tests that were run, but did not pass.

`errors`

The number of tests that IDT couldn't execute.

`disabled`

This attribute is not used and can be ignored.

In the event of test failures or errors, you can identify the test that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag show the test result summary for a test group. For example:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but with a `skipped` attribute that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each executed test for a test group. For example:

```
<testcase classname="Security Test" name="IP Change Tests" attempts="1"></testcase>
```

Attributes used in the `<testcase>` tag

`name`

The name of the test.

`attempts`

The number of times IDT executed the test case.

When a test fails or an error occurs, `<failure>` or `<error>` tags are added to the `<testcase>` tag with information for troubleshooting. For example:

```
<testcase classname="mcu.Full_MQTT" name="MQTT_TestCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
</testcase>
```

Submit IDT usage metrics

If you provide AWS credentials with required permissions, AWS IoT Device Tester collects and submits usage metrics to AWS. This is an opt-in feature and is used to improve IDT functionality. IDT collects information such as the following:

- The AWS account ID used to run IDT
- The IDT CLI commands used to run tests
- The test suite that are run
- The test suites in the *<device-tester-extract-location>* folder
- The number of devices configured in the device pool
- Test case names and run times
- Test result information, such as whether tests passed, failed, encountered errors, or were skipped
- Product features tested
- IDT exit behavior, such as unexpected or early exits

All of the information that IDT sends is also logged to a `metrics.log` file in the *<device-tester-extract-location>*`/results/`*<execution-id>*`/` folder. You can view the log file to see the information that was collected during a test run. This file is generated only if you choose to collect usage metrics.

To disable metrics collection, you do not need to take additional action. Simply do not store your AWS credentials, and if you do have stored AWS credentials, do not configure the `config.json` file to access them.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Provide AWS credentials to IDT

To allow IDT to access your AWS credentials and submit metrics to AWS, do the following:

1. Store the AWS credentials for your IAM user as environment variables or in a credentials file:

   a. To use environment variables, run the following command:

   ```
   AWS_ACCESS_KEY_ID=access-key
   AWS_SECRET_ACCESS_KEY=secret-access-key
   ```

   b. To use the credentials file, add the following information to the `.aws/credentials file:`

   ```
   [profile-name]
   aws_access_key_id=access-key
   aws_secret_access_key=secret-access-key
   ```

2. Configure the `auth` section of the `config.json` file. For more information, see (Optional) Configure config.json.

Maintain test suite versions

IDT for FreeRTOS organizes test resources into test suites and test groups:

- A test suite is the set of test groups used to verify that a device works with particular versions of FreeRTOS.
- A test group is the set of individual tests related to a particular feature, such as BLE and MQTT messaging.

Starting in IDT v3.0.0, test suites are versioned using a `major.minor.patch` format starting from 1.0.0. When you download IDT, the package includes the latest test suite version.

When you start IDT in the command line interface, IDT checks whether a newer test suite version is available. If so, it prompts you to update to the new version. You can choose to update or continue with your current tests.

> **Note**
>
> IDT supports the three latest test suite versions for qualification. For more information, see Understand the support policy for AWS IoT Device Tester.

You can download test suites by using the `upgrade-test-suite` command. Or, you can use the optional parameter `-upgrade-test-suite` *`flag`* when you start IDT where *`flag`* can be 'y' to always download the latest version, or 'n' to use the existing version.

You can also run the `list-supported-versions` command to list the FreeRTOS and test suite versions that are supported by the current version of IDT.

New tests might introduce new IDT configuration settings. If the settings are optional, IDT notifies you and continues running the tests. If the settings are required, IDT notifies you and stops running. After you configure the settings, you can continue to run the tests.

Troubleshoot errors

Each test suite execution has a unique execution ID that is used to create a folder named `results/`*`execution-id`* in the `results` directory. Individual test group logs are under the `results/`*`execution-id`*`/logs` directory. Use the IDT for FreeRTOS console output to find the execution id, test case id, and test group id of the test case that failed and then open the log file for that test case named `results/`*`execution-id`*`/logs/`*`test_group_id`*`__`*`test_case_id`*`.log`. The information in this file includes:

- Full build and flash command output.
- Test execution output.
- More verbose IDT for FreeRTOS console output.

We recommend the following workflow for troubleshooting:

1. If you see the error "*`user/role`* is not authorized to access this resource", make sure that you configure permissions as specified in Create and configure an AWS account.
2. Read the console output to find information, such as execution UUID and currently executing tasks.
3. Look in the `FRQ_Report.xml` file for error statements from each test. This directory contains execution logs of each test group.
4. Look in the log files under `/results/`*`execution-id`*`/logs`.
5. Investigate one of the following problem areas:
   - Device configuration, such as JSON configuration files in the `/configs/` folder.
   - Device interface. Check the logs to determine which interface is failing.

- Device tooling. Make sure that the toolchains for building and flashing the device are installed and configured correctly.
- For FRQ 1.x.x make sure that a clean, cloned version of the FreeRTOS source code is available. FreeRTOS releases are tagged according to the FreeRTOS version. To clone a specific version of the code, use the following commands:

```
git clone --branch version-number https://github.com/aws/amazon-freertos.git
cd amazon-freertos
git submodule update --checkout --init --recursive
```

Troubleshoot device configuration

When you use IDT for FreeRTOS, you must get the correct configuration files in place before you execute the binary. If you're getting parsing and configuration errors, your first step should be to locate and use a configuration template appropriate for your environment. These templates are located in the *IDT_ROOT*`/configs` directory.

If you are still having issues, see the following debugging process.

Where do I look?

Start by reading the console output to find information, such as the execution UUID, which is referenced as `execution-id` in this documentation.

Next, look in the `FRQ_Report.xml` file in the `/results/`*`execution-id`* directory. This file contains all of the test cases that were run and error snippets for each failure. To get all of the execution logs, look for the file `/results/`*`execution-id`*`/logs/`*`test_group_id__test_case_id`*`.log` for each test case.

IDT error codes

The following table explains the error codes generated by IDT for FreeRTOS:

Error Code	Error Code Name	Possible Root Cause	Troubleshooting
201	InvalidInputError	Fields in `device.json` , `config.json` ,	Make sure required fields are not missing

Error Code	Error Code Name	Possible Root Cause	Troubleshooting
		or `userdata.json` are either missing or in an incorrect format.	and are in required format in listed files. For more informati on, see First test of your microcontroller board.

Error Code	Error Code Name	Possible Root Cause	Troubleshooting
202	ValidationError	Fields in `device.js` on , `config.js` on , or `userdata.json` contain invalid values.	Check the error message on the right hand side of the error code in the report: • Invalid AWS Region - Specify a valid AWS region in your `config.js` on file. For more information about AWS regions, see Regions and Endpoints. • Invalid AWS credentials - Set valid AWS credentials on your test machine (through environment variables or the credentials file). Verify that the authentication field is configure d correctly. For more informati on, see Create and configure an AWS account.

Error Code	Error Code Name	Possible Root Cause	Troubleshooting
203	CopySourceCodeError	Unable to copy FreeRTOS source code to specified directory.	Verify the following items: • Check a valid `sourcePath` is specified in your `userdata.json` file. • Delete the `build` folder under FreeRTOS source code directory, if it exists. For more information, see Configure build, flash, and test settings. • Windows has a character limit for file path names. A long file path name will cause an error.

Error Code	Error Code Name	Possible Root Cause	Troubleshooting
204	BuildSourceError	Unable to compile the FreeRTOS source code.	Verify the following items: • Check that the information under `buildTool` in your `userdata.json` file is correct. • If you are using `cmake` as a build tool, make sure the `{{enableTests}}` is specified in the `buildTool` command. For more information, see Configure build, flash, and test settings. • If you have extracted IDT for FreeRTOS to a file path on your system that contains spaces, for example `C:\Users\My Name\Desktop\`, you may need additional quotes inside of your build commands to make

Error Code	Error Code Name	Possible Root Cause	Troubleshooting
			sure the paths are parsed properly. The same thing may be needed for your flash commands.
205	FlashOrRunTestError	IDT FreeRTOS is unable to flash or run FreeRTOS on your DUT.	Verify the informati on under `flashTool` in your `userdata.json` file is correct. For more informati on, see Configure build, flash, and test settings.
206	StartEchoServerError	IDT FreeRTOS is unable to start echo server for the WiFi or secure sockets tests.	Verify the ports configured under `echoServe rConfiguration` in your `userdata.json` file are not in use or blocked by firewall or network settings.

Debug config file parsing errors

Occasionally, a typo in a JSON configuration can lead to parsing errors. Most of the time, the issue is a result of omitting a bracket, comma, or quote from your JSON file. IDT for FreeRTOS performs JSON validation and prints debugging information. It prints the line where the error occurred, the line number, and the column number of the syntax error. This information should be enough to help you fix the error, but if you are still having issues locating the error, you can perform validation manually in your IDE, a text editor such as Atom or Sublime, or through an online tool like JSONLint.

Debug test results parse errors

While running a test group from FreeRTOS-Libraries-Integration-Tests, such as **FullTransportInterfaceTLS, FullPKCS11_Core, FullPKCS11_Onboard_ECC, FullPKCS11_Onboard_RSA, FullPKCS11_PreProvisioned_ECC, FullPKCS11_PreProvisioned_RSA**, or **OTACore**, IDT for FreeRTOS parses the test results from the test device with the serial connection. Sometimes, extra serial outputs on the device may interfere with the parsing of the test results.

In the above mentioned case, strange test case failure reasons like strings originating from unrelated device outputs are output. The IDT for FreeRTOS test case log file (which includes all the serial output IDT for FreeRTOS has received during the test) may show the following:

```
<unrelated device output>
TEST(Full_PKCS11_Capabilities, PKCS11_Capabilities)<unrelated device output>
<unrelated device output>
 PASS
```

In the above example, the unrelated device output prevents IDT for FreeRTOS from detecting the test result which is **PASS**.

Check the following to ensure optimal testing.

- Make sure the logging macros used on the device are thread safe. See Implementing the library logging macros for more information.
- Make sure there are minimal outputs to the serial connection during the tests. Other device outputs can be a problem even if your logging macros are properly thread safe, because the test results will output in separate calls during testing.

An IDT for FreeRTOS test case log would ideally show an uninterrupted test results output like below:

```
---------STARTING TESTS---------
TEST(Full_OTA_PAL, otaPal_CloseFile_ValidSignature) PASS
TEST(Full_OTA_PAL, otaPal_CloseFile_InvalidSignatureBlockWritten) PASS
-----------------------
```

```
2 Tests 0 Failures 0 Ignored
```

Debug integrity check failures

If using FRQ 1.x.x version of FreeRTOS the following integrity checks apply.

When you run the FreeRTOSIntegrity test group and you encounter failures, first make sure that you haven't modified any of the *freertos* directory files. If you haven't, and are still seeing issues, make sure you are using the correct branch. If you run IDT's `list-supported-products` command, you can find which tagged branch of the *freertos* repo you should be using.

If you cloned the correct tagged branch of the `freertos` repo and still have issues, make sure you have also run the `submodule update` command. The clone workflow for the `freertos` repo is as follows.

```
git clone --branch version-number https://github.com/aws/amazon-freertos.git
cd amazon-freertos
git submodule update --checkout —init —recursive
```

The list of files the integrity checker looks for are in the `checksums.json` file in your *freertos* directory. To qualify a FreeRTOS port without any modifications to files and the folder structure, make sure that none of the files listed in the '`exhaustive`' and '`minimal`' sections of the `checksums.json` file have been modified. To run with an SDK configured, verify that none of the files under the '`minimal`' section have been modified.

If you run IDT with an SDK and have modified some files in your *freertos* directory, then make sure you correctly configure your SDK in your `userdata` file. Otherwise, the Integrity checker will verify all files in the *freertos* directory.

Debug FullWiFi test group failures

If you are using FRQ 1.x.x and encounter failures in the FullWiFi test group, and the "`AFQP_WiFiConnectMultipleAP`" test fails, this could be because both access points aren't in the same subnet as the host computer running IDT. Make sure that both access points are in the same subnet as the host computer running IDT.

Debug "required parameter missing" errors

Because new features are being added to IDT for FreeRTOS, changes to the configuration files might be introduced. Using an old configuration file might break your configuration. If this happens, the *test_group_id*__*test_case_id*.log file under the results/*execution-id*/logs directory explicitly lists all missing parameters. IDT for FreeRTOS validates your JSON configuration file schemas to ensure that the latest supported version has been used.

Debug "test could not start" errors

You might see errors that point to failures during test start. Because there are several possible causes, check the following areas for correctness:

- Make sure that the pool name you've included in your execution command actually exists. This is referenced directly from your device.json file.
- Make sure that the device or devices in your pool have correct configuration parameters.

Debug "unable to find start of test results" errors

You might see errors when IDT attempts to parse the results output by the device under test. There are several possible causes, so check the following areas for correctness:

- Make sure that the device under test has a stable connection to your host machine. You can check the log file for a test that shows these errors to see what IDT is receiving.
- If using FRQ 1.x.x, and the device under test is connected via a slow network or other interface, or you do not see the "---------STARTING TESTS---------" flag in a FreeRTOS test group log along with other FreeRTOS test group outputs, you can try increasing the value of testStartDelayms in your userdata configuration. For more information, see Configure build, flash, and test settings.

Debug a "Test failure:expected __ results but saw ___" errors

You might see errors that point to a test failure during testing. The test expects a certain number of results, and does not see it during testing. Some FreeRTOS tests run before IDT sees the output from the device. If you see this error, you can try increasing the value of testStartDelayms in your *userdata* configuration. For more information, see Configure build, flash, and test settings.

Debug a "_____ was unselected due to ConditionalTests constraints" errors

This means that you are running a test on a device pool that is incompatible with the test. This may happen with the OTA E2E tests. For example, while running the `OTADataplaneMQTT` test group and in your `device.json` config file, you have chosen OTA as **No** or `OTADataPlaneProtocol` as **HTTP**. The test group chosen to run must match your `device.json` capability selections.

Debug an IDT timeout during device output monitoring

IDT can timeout due to a number of reasons. If a timeout happens during the device output monitoring phase of a test, and you can see the results inside of the IDT test case log, it means that the results were incorrectly parsed by IDT. One reason could be the interleaved log messages in the middle of the test results. If this is the case, please refer to the FreeRTOS Porting Guide for further details on how the UNITY logs should be setup.

Another reason for a timeout during device output monitoring could be a device rebooting after a single TLS test case failure. The device then runs the flashed image and causes an infinite loop which is seen in the logs. If this happens, make sure your device does not reboot after a test failure.

Debug a "not authorized to access resource" error

You might see the error "*`user/role`* is not authorized to access this resource" in the terminal output or in the `test_manager.log` file under `/results/`*`execution-id`*`/logs`. To resolve this issue, attach the `AWSIoTDeviceTesterForFreeRTOSFullAccess` managed policy to your test user. For more information, see Create and configure an AWS account.

Debug network test errors

For network-based tests, IDT starts an echo server that binds to a non-reserved port on the host machine. If you are running into errors due to timeouts or unavailable connections in the WiFi or secure sockets tests, make sure that your network is configured to allow traffic to configured ports in the 1024 - 49151 range.

The secure sockets test uses ports 33333 and 33334 by default. The WiFi tests uses port 33335 by default. If these three ports are in use or blocked by firewall or network, you can choose to use different ports in userdata.json for testing. For more information, see Configure build, flash, and test settings. You can use the following commands to check whether a specific port is in use:

- Windows: `netsh advfirewall firewall show rule name=all | grep port`

- Linux: `sudo netstat -pan | grep port`
- macOS: `netstat -nat | grep port`

OTA update failures due to same version payload

If OTA test cases are failing due to the same version being on the device after an OTA was performed, it may be due to your build system (e.g. cmake) not noticing IDT's changes to the FreeRTOS source code and not building an updated binary. This causes OTA to be performed with the same binary that is currently on the device, and the test to fail. To troubleshoot OTA update failures, start by making sure that you are using the latest supported version of your build system.

OTA test failure on `PresignedUrlExpired` test case

One prerequisite of this test is that the OTA update time should be more than 60 seconds, otherwise the test would fail. If this occurs, the following error message is found in the log: "Test takes less than 60 seconds (url expired time) to finish. Please reach out to us."

Debug device interface and port errors

This section contains information about the device interfaces IDT uses to connect to your devices.

Supported platforms

IDT supports Linux, macOS, and Windows. All three platforms have different naming schemes for serial devices that are attached to them:

- Linux: `/dev/tty*`
- macOS: `/dev/tty.*` or `/dev/cu.*`
- Windows: COM*

To check your device port:

- For Linux/macOS, open a terminal and run `ls /dev/tty*`.
- For macOS, open a terminal and run `ls /dev/tty.*` or `ls /dev/cu.*`.
- For Windows, open Device Manager and expand the serial devices group.

To verify which device is connected to a port:

- For Linux, make sure that the `udev` package is installed, and then run `udevadm info --name=PORT`. This utility prints the device driver information that helps you verify you are using the correct port.
- For macOS, open Launchpad and search for **`System Information`**.
- For Windows, open Device Manager and expand the serial devices group.

Device interfaces

Each embedded device is different, which means that they can have one or more serial ports. It is common for devices to have two ports when connected to a machine:

- A data port for flashing the device.
- A read port to read output.

  You must set the correct read port in your `device.json` file. Otherwise, reading output from the device might fail.

  In the case of multiple ports, make sure to use the read port of the device in your `device.json` file. For example, if you plug in an Espressif WRover device and the two ports assigned to it are `/dev/ttyUSB0` and `/dev/ttyUSB1`, use `/dev/ttyUSB1` in your `device.json` file.

For Windows, follow the same logic.

Reading device data

IDT for FreeRTOS uses individual device build and flash tooling to specify port configuration. If you are testing your device and don't get output, try the following default settings:

- Baud rate: 115200
- Data bits: 8
- Parity: None
- Stop bits: 1
- Flow control: None

These settings are handled by IDT for FreeRTOS. You do not have to set them. However, you can use the same method to manually read device output. On Linux or macOS, you can do this with the `screen` command. On Windows, you can use a program such as TeraTerm.

`Screen: screen /dev/cu.usbserial 115200`

`TeraTerm: Use the above-provided settings to set the fields explicitly in the GUI.`

Development toolchain problems

This section discusses problems that can occur with your toolchain.

Code Composer Studio on Ubuntu

Newer versions of Ubuntu (17.10 and 18.04) have a version of the `glibc` package that is not compatible with Code Composer Studio 7.*x* versions. We recommended that you install Code Composer Studio version 8.2 or later.

Symptoms of incompatibility might include:

- FreeRTOS failing to build or flash to your device.
- The Code Composer Studio installer might freeze.
- No log output is displayed in the console during the build or flash process.
- Build command attempts to launch in GUI mode even when invoked as headless.

Logging

IDT for FreeRTOS logs are placed in a single location. From the root IDT directory, these files are available under `results/`*`execution-id`*`/`:

- `FRQ_Report.xml`
- `awsiotdevicetester_report.xml`
- `logs/`*`test_group_id`*`__`*`test_case_id`*`.log`

`FRQ_Report.xml` and `logs/`*`test_group_id`*`__`*`test_case_id`*`.log` are the most important logs to examine. `FRQ_Report.xml` contains information about which test cases failed with a specific error message. You can then use `logs/`*`test_group_id`*`__`*`test_case_id`*`.log` to dig further into the problem to get better context.

Console errors

When AWS IoT Device Tester is run, failures are reported to the console with brief messages. Look in `results/`*`execution-id`*`/logs/`*`test_group_id`*`__`*`test_case_id`*`.log` to learn more about the error.

Log errors

Each test suite execution has a unique execution ID that is used to create a folder named `results/`*`execution-id`*. Individual test case logs are under the `results/`*`execution-id`*`/logs` directory. Use the output of the IDT for FreeRTOS console to find the execution id, test case id, and test group id of the test case that failed. Then use this information to find and open the log file for that test case named `results/`*`execution-id`*`/logs/`*`test_group_id`*`__`*`test_case_id`*`.log` The information in this file includes the full build and flash command output, test execution output, and more verbose AWS IoT Device Tester console output.

S3 bucket issues

If you press **CTRL+C** while running IDT, IDT will start a clean up process. Part of that clean up is to remove Amazon S3 resources that have been created as a part of the IDT tests. If the clean up can't finish, you might run into an issue where you have too many Amazon S3 buckets that have been created. This means the next time that you run IDT the tests will start to fail.

If you press **CTRL+C** to stop IDT, you must let it finish the clean up process to avoid this issue. You can also delete the Amazon S3 buckets from your account that were created manually.

Troubleshoot timeout errors

If you see timeout errors while running a test suite, increase the timeout by specifying a timeout multiplier factor. This factor is applied to the default timeout value. Any value configured for this flag must be greater than or equal to 1.0. To use the timeout multiplier, use the flag `--timeout-multiplier` when running the test suite.

Example

IDT v3.0.0 and later

```
./devicetester_linux run-suite --suite-id FRQ_1.0.1 --pool-id DevicePool1 --timeout-multiplier 2.5
```

IDT v1.7.0 and earlier

```
./devicetester_linux run-suite --suite-id FRQ_1 --pool-id DevicePool1 --timeout-
multiplier 2.5
```

Cellular feature and AWS charges

When the `Cellular` feature is set to `Yes` in your `device.JSON` file, FullSecureSockets will use t.micro EC2 instances for running tests and this may incur additional costs to your AWS account. For more information, see Amazon EC2 pricing.

Qualification report generation policy

Qualification reports are only generated by AWS IoT Device Tester (IDT) versions that support FreeRTOS versions released within the last two years. If you have questions about the support policy, please contact AWS Support.

Understand the AWS Managed policy for AWS IoT Device Tester

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

Topics

- AWS managed policy: AWSIoTDeviceTesterForFreeRTOSFullAccess

- Updates to AWS managed policies

AWS managed policy: AWSIoTDeviceTesterForFreeRTOSFullAccess

The `AWSIoTDeviceTesterForFreeRTOSFullAccess` managed policy contains the following AWS IoT Device Tester permissions for version checking, auto update features, and collection of metrics.

Permission details

This policy includes the following permissions:

- `iot-device-tester:SupportedVersion`

  Grants AWS IoT Device Tester permission to fetch the list of supported products, test suites and IDT versions.
- `iot-device-tester:LatestIdt`

  Grants AWS IoT Device Tester permission to fetch the latest IDT version available for download.
- `iot-device-tester:CheckVersion`

  Grants AWS IoT Device Tester permission to check version compatibility for IDT, test suites and products.
- `iot-device-tester:DownloadTestSuite`

  Grants AWS IoT Device Tester permission to download test suite updates.
- `iot-device-tester:SendMetrics`

  Grants AWS permission to collect metrics about AWS IoT Device Tester internal usage.

JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
```

```
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::*:role/idt-*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "iot.amazonaws.com"
                }
            }
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "iot:DeleteThing",
                "iot:AttachThingPrincipal",
                "iot:DeleteCertificate",
                "iot:GetRegistrationCode",
                "iot:CreatePolicy",
                "iot:UpdateCACertificate",
                "s3:ListBucket",
                "iot:DescribeEndpoint",
                "iot:CreateOTAUpdate",
                "iot:CreateStream",
                "signer:ListSigningJobs",
                "acm:ListCertificates",
                "iot:CreateKeysAndCertificate",
                "iot:UpdateCertificate",
                "iot:CreateCertificateFromCsr",
                "iot:DetachThingPrincipal",
                "iot:RegisterCACertificate",
                "iot:CreateThing",
                "iam:ListRoles",
                "iot:RegisterCertificate",
                "iot:DeleteCACertificate",
                "signer:PutSigningProfile",
                "s3:ListAllMyBuckets",
                "signer:ListSigningPlatforms",
                "iot-device-tester:SendMetrics",
                "iot-device-tester:SupportedVersion",
                "iot-device-tester:LatestIdt",
                "iot-device-tester:CheckVersion",
                "iot-device-tester:DownloadTestSuite"
            ],
            "Resource": "*"
        },
```

```
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "signer:StartSigningJob",
                "acm:GetCertificate",
                "signer:DescribeSigningJob",
                "s3:CreateBucket",
                "execute-api:Invoke",
                "s3:DeleteBucket",
                "s3:PutBucketVersioning",
                "signer:CancelSigningProfile"
            ],
            "Resource": [
                "arn:aws:execute-api:us-east-1:098862408343:9xpmnvs5h4/prod/POST/
metrics",
                "arn:aws:signer:*:*:/signing-profiles/*",
                "arn:aws:signer:*:*:/signing-jobs/*",
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:acm:*:*:certificate/*",
                "arn:aws:s3:::idt-*",
                "arn:aws:s3:::afr-ota*"
            ]
        },
        {
            "Sid": "VisualEditor3",
            "Effect": "Allow",
            "Action": [
                "iot:DeleteStream",
                "iot:DeleteCertificate",
                "iot:AttachPolicy",
                "iot:DetachPolicy",
                "iot:DeletePolicy",
                "s3:ListBucketVersions",
                "iot:UpdateCertificate",
                "iot:GetOTAUpdate",
                "iot:DeleteOTAUpdate",
                "iot:DescribeJobExecution"
            ],
            "Resource": [
                "arn:aws:s3:::afr-ota*",
                "arn:aws:iot:*:*:thinggroup/idt*",
                "arn:aws:iam::*:role/idt-*"
```

```
            ]
        },
        {
            "Sid": "VisualEditor4",
            "Effect": "Allow",
            "Action": [
                "iot:DeleteCertificate",
                "iot:AttachPolicy",
                "iot:DetachPolicy",
                "s3:DeleteObjectVersion",
                "iot:DeleteOTAUpdate",
                "s3:PutObject",
                "s3:GetObject",
                "iot:DeleteStream",
                "iot:DeletePolicy",
                "s3:DeleteObject",
                "iot:UpdateCertificate",
                "iot:GetOTAUpdate",
                "s3:GetObjectVersion",
                "iot:DescribeJobExecution"
            ],
            "Resource": [
                "arn:aws:s3:::afr-ota*/*",
                "arn:aws:s3:::idt-*/*",
                "arn:aws:iot:*:*:policy/idt*",
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:iot:*:*:otaupdate/idt*",
                "arn:aws:iot:*:*:thing/idt*",
                "arn:aws:iot:*:*:cert/*",
                "arn:aws:iot:*:*:job/*",
                "arn:aws:iot:*:*:stream/*"
            ]
        },
        {
            "Sid": "VisualEditor5",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::afr-ota*/*",
                "arn:aws:s3:::idt-*/*"
            ]
```

```
        },
        {
            "Sid": "VisualEditor6",
            "Effect": "Allow",
            "Action": [
                "iot:CancelJobExecution"
            ],
            "Resource": [
                "arn:aws:iot:*:*:job/*",
                "arn:aws:iot:*:*:thing/idt*"
            ]
        },
        {
            "Sid": "VisualEditor7",
            "Effect": "Allow",
            "Action": [
                "ec2:TerminateInstances"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:instance/*"
            ],
            "Condition": {
                "StringEquals": {
                    "ec2:ResourceTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
            "Sid": "VisualEditor8",
            "Effect": "Allow",
            "Action": [
                "ec2:AuthorizeSecurityGroupIngress",
                "ec2:DeleteSecurityGroup"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:security-group/*"
            ],
            "Condition": {
                "StringEquals": {
                    "ec2:ResourceTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
```

```
            "Sid": "VisualEditor9",
            "Effect": "Allow",
            "Action": [
                "ec2:RunInstances"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:instance/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
            "Sid": "VisualEditor10",
            "Effect": "Allow",
            "Action": [
                "ec2:RunInstances"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:image/*",
                "arn:aws:ec2:*:*:security-group/*",
                "arn:aws:ec2:*:*:volume/*",
                "arn:aws:ec2:*:*:key-pair/*",
                "arn:aws:ec2:*:*:placement-group/*",
                "arn:aws:ec2:*:*:snapshot/*",
                "arn:aws:ec2:*:*:network-interface/*",
                "arn:aws:ec2:*:*:subnet/*"
            ]
        },
        {
            "Sid": "VisualEditor11",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateSecurityGroup"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:security-group/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "IoTDeviceTester"
                }
```

```
            }
        },
        {
            "Sid": "VisualEditor12",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:DescribeSecurityGroups",
                "ssm:DescribeParameters",
                "ssm:GetParameters"
            ],
            "Resource": "*"
        },
        {
            "Sid": "VisualEditor13",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateTags"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:security-group/*",
                "arn:aws:ec2:*:*:instance/*"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "aws:TagKeys": [
                        "Owner"
                    ]
                },
                "StringEquals": {
                    "ec2:CreateAction": [
                        "RunInstances",
                        "CreateSecurityGroup"
                    ]
                }
            }
        }
    ]
}
```

Updates to AWS managed policies

You can view details about updates to AWS managed policies for AWS IoT Device Tester from the time this service began tracking these changes.

Version	Change	Description	Date
7 (Latest)	Restructured the `ec2:CreateTags` conditions.	Removing usage of `ForAnyValues` .	6/14/2023
6	Removed `freertos:ListHardwarePlatforms` from the policy.	Removing permissions as this action is deprecated as of March 1st, 2023.	6/2/2023
5	Added permissions to run echo server tests using EC2.	This is for starting and stopping an EC2 instance in customers ' AWS accounts.	12/15/2020
4	Added `iot:CancelJobExecution` .	This permission cancels OTA jobs.	7/17/2020
3	Added the following permissions: • `iot-device-tester:DownloadTestSuite` , • `iot-device-tester:CheckVersion` , • `iot-device-tester:LatestIdt` ,	• `iot-device-tester:DownloadTestSuite` — Grants AWS IoT Device Tester permission to download test suite updates, • `iot-device-tester:CheckVersion` — Grants AWS IoT Device	3/23/2020

Version	Change	Description	Date
	• `iot-device-tester: SupportedVersion` .	Tester permission to check version compatibility for IDT, test suites and products, • `iot-device-tester: LatestIdt` — Grants AWS IoT Device Tester permission to fetch the latest IDT version available for download, • `iot-device-tester: SupportedVersion` — Grants AWS IoT Device Tester permission to fetch the list of supported products, test suites and IDT versions.	
2	Added `iot-device-tester: SendMetrics` permissions.	Grants AWS permission to collect metrics about AWS IoT Device Tester internal usage.	2/18/2020
1	Initial version.		2/12/2020

Understand the support policy for AWS IoT Device Tester

Important

As of October 2022, AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 does not generate signed qualification reports. You cannot qualify new AWS IoT FreeRTOS devices to list in the AWS Partner Device Catalog through the AWS Device Qualification Program using IDT FRQ 1.0 versions. While you can't qualify FreeRTOS devices using IDT FRQ 1.0, you can continue to test your FreeRTOS devices with FRQ 1.0. We recommend that you use IDT FRQ 2.0 to qualify and list FreeRTOS devices in the AWS Partner Device Catalog.

AWS IoT Device Tester for FreeRTOS is a test automation tool to validate FreeRTOS port to devices. Additionally you can qualify your FreeRTOS devices and list them on the AWS Partner Device Catalog . The AWS IoT Device Tester for FreeRTOS supports validation and qualification of FreeRTOS Long Term Supported (LTS) libraries available on GitHub at FreeRTOS/FreeRTOS-LTS , and FreeRTOS mainline available at FreeRTOS/FreeRTOS. We recommend that you use the most recent versions of both the FreeRTOS and AWS IoT Device Tester for FreeRTOS to validate and qualify your devices.

For FreeRTOS-LTS, IDT supports the validation and qualification of FreeRTOS 202210 LTS version. See here for more information on FreeRTOS LTS releases and their maintenance time line. Once the support period of these LTS releases ends, you can still continue validation, but IDT will not generate a report, that will allow you to submit your device for qualification.

For the mainline FreeRTOS available at FreeRTOS/FreeRTOS , we support the validation and qualification of all the versions released in the past six months, or the previous two versions of FreeRTOS if released more than six months apart. See here for currently supported versions. For unsupported versions of FreeRTOS, you can still continue validation, but IDT will not generate a report, that will allow you to submit your device for qualification.

See Supported versions of AWS IoT Device Tester for the latest supported IDT and FreeRTOS versions. You can use any of the supported versions of AWS IoT Device Tester with the corresponding version of FreeRTOS to test or qualify your device. If you continue to use the Unsupported IDT versions for FreeRTOS, you will not receive the latest bug fixes or updates.

For questions about the support policy, contact AWS Customer Support.

Security in AWS

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to an AWS service, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using AWS. The following topics show you how to configure AWS to meet your security and compliance objectives. You'll also learn how to use AWS services that can help you to monitor and secure your AWS resources.

For more in-depth information about AWS IoT security see Security and Identity for AWS IoT.

Topics

- Identity and Access Management for FreeRTOS
- Compliance validation
- Resilience in AWS
- Infrastructure security in FreeRTOS

Identity and Access Management for FreeRTOS

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in)

and *authorized* (have permissions) to use FreeRTOS resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How FreeRTOS works with IAM
- Identity-based policy examples for FreeRTOS
- Troubleshoot FreeRTOS identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in FreeRTOS.

Service user – If you use the FreeRTOS service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more FreeRTOS features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in FreeRTOS, see Troubleshoot FreeRTOS identity and access.

Service administrator – If you're in charge of FreeRTOS resources at your company, you probably have full access to FreeRTOS. It's your job to determine which FreeRTOS features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with FreeRTOS, see How FreeRTOS works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to FreeRTOS. To view example FreeRTOS identity-based policies that you can use in IAM, see Identity-based policy examples for FreeRTOS.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a

role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can

perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user

or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How FreeRTOS works with IAM

Before you use IAM to manage access to FreeRTOS, learn what IAM features are available to use with FreeRTOS.

IAM features you can use with FreeRTOS

IAM feature	FreeRTOS support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how FreeRTOS and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for FreeRTOS

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all

of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for FreeRTOS

To view examples of FreeRTOS identity-based policies, see Identity-based policy examples for FreeRTOS.

Resource-based policies within FreeRTOS

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for FreeRTOS

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API

operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of FreeRTOS actions, see Actions defined by FreeRTOS in the *Service Authorization Reference*.

Policy actions in FreeRTOS use the following prefix before the action:

```
awes
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "awes:action1",
      "awes:action2"
         ]
```

To view examples of FreeRTOS identity-based policies, see Identity-based policy examples for FreeRTOS.

Policy resources for FreeRTOS

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of FreeRTOS resource types and their ARNs, see Resources defined by FreeRTOS in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by FreeRTOS.

To view examples of FreeRTOS identity-based policies, see Identity-based policy examples for FreeRTOS.

Policy condition keys for FreeRTOS

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of FreeRTOS condition keys, see Condition keys for FreeRTOS in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by FreeRTOS.

To view examples of FreeRTOS identity-based policies, see Identity-based policy examples for FreeRTOS.

ACLs in FreeRTOS

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with FreeRTOS

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with FreeRTOS

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your

company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switch from a user to an IAM role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for FreeRTOS

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for FreeRTOS

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break FreeRTOS functionality. Edit service roles only when FreeRTOS provides guidance to do so.

Service-linked roles for FreeRTOS

Supports service-linked roles: No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for FreeRTOS

By default, users and roles don't have permission to create or modify FreeRTOS resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by FreeRTOS, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for FreeRTOS in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the FreeRTOS console
- Allow users to view their own permissions

Policy best practices

Identity-based policies determine whether someone can create, access, or delete FreeRTOS resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We

recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.

- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the FreeRTOS console

To access the FreeRTOS console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the FreeRTOS resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the FreeRTOS console, also attach the FreeRTOS *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Troubleshoot FreeRTOS identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with FreeRTOS and IAM.

Topics

- I am not authorized to perform an action in FreeRTOS
- I am not authorized to perform iam:PassRole
- Allow people outside of my AWS account to access my FreeRTOS resources

I am not authorized to perform an action in FreeRTOS

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `awes:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `awes:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to FreeRTOS.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in FreeRTOS. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

Allow people outside of my AWS account to access my FreeRTOS resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether FreeRTOS supports these features, see How FreeRTOS works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance validation

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in AWS

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with

low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Infrastructure security in FreeRTOS

AWS managed services are protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access AWS services through the network. Clients must support Transport Layer Security (TLS) 1.2 or later. We recommend TLS 1.3 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Amazon-FreeRTOS Github Repository Migration Guide

If you have an existing FreeRTOS project based on the now deprecated amazon-freertos repository, follow these steps:

1. Stay up to date with the latest, publicly available security fixes. Check the FreeRTOS LTS libraries page for updates, or subscribe to the FreeRTOS-LTS GitHub repository to receive the latest LTS patches with critical and security bug fixes. You can download or clone the latest FreeRTOS LTS patches required directly from the individual GitHub repositories.
2. Consider refactoring the network transport interface implementation to optimize your hardware platform. The abstract APIs like secure sockets and Wifi APIs are not required by the latest coreMQTT library. See Transport Interface for further details.

Appendix

The following table provides recommendations for all demo projects, legacy libraries, and abstract APIs within the Amazon-FreeRTOS repository.

Migrated libraries and demos

Name	Type	Recommendations	
coreHTTP	demos and library	Clone or download the coreHTTP library directly from the coreHTTP repository (sub-module if using git) in the FreeRTOS Github organization. The coreHTTP demos are in the primary FreeRTOS distribution. For more details, refer to the coreHTTP page.	
coreMQTT	demos and library	Clone or download the coreMQTT library directly from the coreMQTT repository (sub-module if using git) in the FreeRTOS Github organization. The coreMQTT demos are in the primary FreeRTOS distribution. For	

Name	Type	Recommendations
		more details, refer to the coreMQTT page.
coreMQTT-Agent	demos and library	Clone or download the coreMQTT-Agent library directly from the coreMQTT-Agent repository (sub-module if using git) in the FreeRTOS Github organization. The coreMQTT-Agent demos are in the coreMQTT-Agent-Demos repository. For more details, refer to the coreMQTT-Agent page.
device_defender_for_aws	demos and library	The AWS IoT Device Defender library is in its repository in the AWS GitHub organisation. Clone or download it (sub-module if using git) directly from the AWS IoT Device Defender repository. The AWS IoT Device Defender demos are in the primary FreeRTOS distribution. For more details, refer to the AWS IoT Device Defender page.
device_shadow_for_aws	demos and library	The AWS IoT Device Shadow library is in its repository in the AWS GitHub organisation. Clone or download it (sub-module if using git) directly from the AWS IoT Device Shadow) repository. The AWS IoT Device Shadow demos are in the primary FreeRTOS distribution. For more details, refer to the AWS IoT Device Shadow page.

Name	Type	Recommendations
jobs_for_aws	demos and library	The AWS IoT Jobs library is in its repository in the AWS GitHub organization. Clone or download it (sub-module if using git) directly from the AWS IoT Jobs repository. The AWS IoT Jobs demos are in the primary FreeRTOS distribution. For more details, refer to the AWS IoT Jobs page.
OTA	demos and library	The AWS IoT Over-The-Air (OTA) Update library is in its repository in the AWS GitHub organization. Clone or download it (sub-module if using git) directly from the AWS IoT OTA repository. The AWS IoT OTA demos are in the primary FreeRTOS distribution. For more details, refer to the AWS IoT OTA page.
CLI and FreeRTOS_ Plus_CLI	demos and library	There is a CLI example running on WinSim. Refer to the FreeRTOS Plus Command Line Interface page for more details. The Featured FreeRTOS IoT reference integrations on the NXP i.MX RT1060 and STM32U5 platforms, also provide CLI examples on actual hardware.

Name	Type	Recommendations
logging	macro	There are implementations of the logging macro for specific hardware platforms used by some of the FreeRTOS libraries. Refer to the logging page for how to implement the logging macro. Refer to one of the FreeRTOS featured IoT references for an example running on actual hardware.
greengrass_connectivity	demo	[Migration in progress] This demo project assumed that cloud connectivity was available before connecting to an AWS IoT Greengrass device. A new project that demonstrates local authentication and discovery capability is under development. Expect the new demo project to be published shortly in the FreeRTOS Github organization.

Deprecated libraries and demos

Name	Type	Recommendations
BLE	demos and libraries	The FreeRTOS BLE library implements the proprietary MQTT protocol and supports publishing and subscribing to MQTT topics over Bluetooth Low Energy (BLE) through a proxy device such as a mobile phone. This is no longer mandated. Use either your own BLE stack or a

Name	Type	Recommendations
		third-party option such as NimBLE to best optimize your project.
dev_mode_key_provisioning	demos	The Featured FreeRTOS IoT reference integrations on the NXP i.MX RT1060, STM32U5, or ESP32-C3 platforms provide examples of crucial provisioning using a CLI.
posix	abstraction and demo	Not recommended for use.
wifi_provisioning	example	This example demonstrated how to provision WiFi credentials on a device using the Amazon-FreeRTOS BLE library. Refer to the FreeRTOS Featured IoT reference on the ESP32C3 platform for an example of WiFi provisioning via BLE.
Legacy abstract APIs	code	These are APIs that were created to provide an abstract interface for various third-party software stacks, connectivity modules, and MCU platforms from a variety of vendors. For example, there are interfaces for WiFi abstraction, secure sockets, and so on. They are supported in the Amazon-FreeRTOS repository and are in the folder `/libraries/abstractions/` . These APIs are not required when using the FreeRTOS LTS libraries.

The libraries and demos in the table above will not get security patches or bug fixes.

Third-party libraries

When demos in Amazon-FreeRTOS use third-party libraries, we recommend that you sub-module them directly from their third-party repositories.

- **CMock**: clone it (submodule if you use git) directly from the Cmock repository.
- **jsmn**: not recommended and no longer supported.
- **lwip**: clone it (submodule if you use git) directly from the lwip-tcpip repository.
- **lwip_osal**: refer to the FreeRTOS Featured Reference Integrations on i.MX RT1060 or STM32U5 for how to implement lwip_osal on your hardware platform/board.
- **mbedtls**: clone it (submodule if you use git) directly from the Mbed-TLS repository. The mbedtls config and utilities can be reused; make a local copy in this case.
- **pkcs11**: clone it (submodule if you use git) directly from either the corePKCS11 library or the OASIS PKCS 11 repository.
- **tinycbor**: clone it (submodule if you use git) directly from thetinycbor repository.
- **tinycrypt**: we recommend that you use crypto accelerators from your MCU platform, if available. If you want to continue to use tinycrypt, clone it (submodule if you use git) directly from the tinycrypt repository.
- **tracealyzer_recorder**: clone it (submodule if you use git) directly from Percepio's trace recorder repository.
- **unity**: clone it (submodule if you use git) directly from the ThrowTheSwitch/Unity repository.
- **win_pcap**: win_pcap is no longer maintained. We recommend that you use libslirp, libpcap (posix), or npcap instead.

Porting tests and integration tests

All tests under the `/tests` folder that are required to validate integration of FreeRTOS libraries were migrated to the FreeRTOS-Libraries-Integration-Tests repository. These can be used to test PAL implementation and library integration. The same tests are used by AWS IoT Device Tester (IDT) for the AWS Device Qualification Program for FreeRTOS.

FreeRTOS Archived documentation

This content is archived and no longer updated. It can reference outdated information or deprecated features. For the latest guidance on FreeRTOS, please refer to the most current information in this FreeRTOS User Guide or other relevant AWS documentation. For current information, see What is FreeRTOS?.

Topics

- FreeRTOS User Guide Archive
- Previous FreeRTOS User Guide contents

FreeRTOS User Guide Archive

These previous versions of the FreeRTOS User Guide are available for use with FreeRTOS LTS (long term support) releases.

- FreeRTOS User Guide for FreeRTOS version 202210.00
- FreeRTOS User Guide for FreeRTOS version 202012.00

Previous FreeRTOS User Guide contents

This content is obsolete but provided here for reference.

See Get started with FreeRTOS for links to recent content.

Get Started with FreeRTOS

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This Getting Started with FreeRTOS tutorial shows you how to download and configure FreeRTOS on a host machine, and then compile and run a simple demo application on a qualified microcontroller board.

Throughout this tutorial, we assume that you are familiar with AWS IoT and the AWS IoT console. If not, we recommend that you complete the AWS IoT Getting Started tutorial before you continue.

Topics:

- FreeRTOS demo application
- First steps
- Troubleshooting getting started
- Using CMake with FreeRTOS
- Developer-mode key provisioning
- Board-specific getting started guides
- Next steps with FreeRTOS

FreeRTOS demo application

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The demo application in this tutorial is the coreMQTT Agent demo defined in the *freertos*/demos/coreMQTT_Agent/mqtt_agent_task.c file. It uses the coreMQTT library to connect to the AWS Cloud and then periodically publish messages to an MQTT topic hosted by the AWS IoT MQTT broker.

Only a single FreeRTOS demo application can run at a time. When you build a FreeRTOS demo project, the first demo enabled in the *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h header file is the application that runs. For this tutorial, you do not need to enable or disable any demos. The coreMQTT Agent demo is enabled by default.

For more information about the demo applications included with FreeRTOS, see FreeRTOS demos.

First steps

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

To get started using FreeRTOS with AWS IoT, you must have an AWS account, a user with permissions to access AWS IoT and FreeRTOS cloud services. You also must download FreeRTOS and configure your board's FreeRTOS demo project to work with AWS IoT. The following sections walk you through these requirements.

> **Note**
>
> - If you're using the Espressif ESP32-DevKitC, ESP-WROVER-KIT, or the ESP32-WROOM-32SE, skip these steps and go to Getting started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT.
> - If you're using the Nordic nRF52840-DK, skip these steps and go to Getting started with the Nordic nRF52840-DK.

1. Setting up your AWS account and permissions
2. Registering your MCU board with AWS IoT
3. Downloading FreeRTOS
4. Configuring the FreeRTOS demos

Setting up your AWS account and permissions

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:

- Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
- (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Registering your MCU board with AWS IoT

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

An AWS IoT policy

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

An AWS IoT thing

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

A private key and X.509 certificate

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

To register your board, follow the procedures below.

To create an AWS IoT policy

1. To create an IAM policy, you must know your AWS Region and AWS account number.

   To find your AWS account number, open the AWS Management Console, locate and expand the menu beneath your account name in the upper-right corner, and choose **My Account**. Your account ID is displayed under **Account Settings**.

   To find the AWS region for your AWS account, use the AWS Command Line Interface. To install the AWS CLI, follow the instructions in the AWS Command Line Interface User Guide. After you install the AWS CLI, open a command prompt window and enter the following command:

   ```
   aws iot describe-endpoint --endpoint-type=iot:Data-ATS
   ```

   The output should look like this:

```
{
    "endpointAddress": "xxxxxxxxxxxxx-ats.iot.us-west-2.amazonaws.com"
}
```

In this example, the region is `us-west-2`.

> **Note**
>
> We recommend using ATS endpoints as seen in the example.

2. Browse to the AWS IoT console.
3. In the navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**.
4. Enter a name to identify your policy.
5. In the **Add statements** section, choose **Advanced mode**. Copy and paste the following JSON into the policy editor window. Replace *aws-region* and *aws-account* with your AWS Region and account ID.

   JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
```

```
            "Action": "iot:Receive",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        }
    ]
}
```

This policy grants the following permissions:

`iot:Connect`

Grants your device the permission to connect to the AWS IoT message broker with any client ID.

`iot:Publish`

Grants your device the permission to publish an MQTT message on any MQTT topic.

`iot:Subscribe`

Grants your device the permission to subscribe to any MQTT topic filter.

`iot:Receive`

Grants your device the permission to receive messages from the AWS IoT message broker on any MQTT topic.

6. Choose **Create**.

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

1. Browse to the AWS IoT console.
2. In the navigation pane, choose **Manage**, and then choose **Things**.
3. If you do not have any IoT things registered in your account, the **You don't have any things yet** page is displayed. If you see this page, choose **Register a thing**. Otherwise, choose **Create**.
4. On the **Creating AWS IoT things** page, choose **Create a single thing**.
5. On the **Add your device to the thing registry** page, 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 choose **Register thing**.

After your board is registered with AWS IoT, you can continue to Downloading FreeRTOS.

Downloading FreeRTOS

You can download FreeRTOS from the FreeRTOS GitHub repository.

After you download FreeRTOS, you can continue to Configuring the FreeRTOS demos.

Configuring the FreeRTOS demos

You must edit some configuration files in your FreeRTOS directory before you can compile and run any demos on your board.

To configure your AWS IoT endpoint

You must provide FreeRTOS with your AWS IoT endpoint so the application running on your board can send requests to the correct endpoint.

1. Browse to the AWS IoT console.
2. In the left navigation pane, choose **Settings**.

   Your AWS IoT endpoint is displayed in **Device data 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. Open `demos/include/aws_clientcredential.h`.
5. Specify values for the following constants:

   - `#define clientcredentialMQTT_BROKER_ENDPOINT "`*Your AWS IoT endpoint*`";`
   - `#define clientcredentialIOT_THING_NAME "`*The AWS IoT thing name of your board*`"`

To configure your Wi-Fi

If your board is connecting to the internet across a Wi-Fi connection, you must provide FreeRTOS with Wi-Fi credentials to connect to the network. If your board does not support Wi-Fi, you can skip these steps.

1. `demos/include/aws_clientcredential.h`.
2. Specify values for the following `#define` constants:

   - `#define clientcredentialWIFI_SSID "`*`The SSID for your Wi-Fi network`*`"`
   - `#define clientcredentialWIFI_PASSWORD "`*`The password for your Wi-Fi network`*`"`
   - `#define clientcredentialWIFI_SECURITY` *`The security type of your Wi-Fi network`*

     Valid security types are:

     - `eWiFiSecurityOpen` (Open, no security)
     - `eWiFiSecurityWEP` (WEP security)
     - `eWiFiSecurityWPA` (WPA security)
     - `eWiFiSecurityWPA2` (WPA2 security)

To format your AWS IoT credentials

FreeRTOS must have the AWS IoT certificate and private keys associated with your registered thing and its permissions policies to successfully communicate with AWS IoT on behalf of your device.

> **Note**
>
> To configure your AWS IoT credentials, you must have the private key and certificate that you downloaded from the AWS IoT console when you registered your device. After you have registered your device as an AWS IoT thing, you can retrieve device certificates from the AWS IoT console, but you cannot retrieve private keys.

FreeRTOS is a C language project, and the certificate and private key must be specially formatted to be added to the project.

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

After you configure FreeRTOS, you can continue to the Getting Started guide for your board to set up your platform's hardware and its software development environment, and then compile and run the demo on your board. For board-specific instructions, see the Board-specific getting started guides. The demo application that is used in the Getting Started tutorial is the coreMQTT Mutual Authentication demo, which is located at `demos/coreMQTT/mqtt_demo_mutual_auth.c`.

Troubleshooting getting started

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The following topics can help you troubleshoot issues that you encounter while getting started with FreeRTOS:

Topics

- General getting started troubleshooting tips
- Installing a terminal emulator

For board-specific troubleshooting, see the Get Started with FreeRTOS guide for your board.

General getting started troubleshooting tips

No messages appear in the AWS IoT console after I run the Hello World demo project. What do I do?

Try the following:

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

The logs shown in my terminal when running a demo are truncated. How can I increase their length?

Increase the value of `configLOGGING_MAX_MESSAGE_LENGTH` to 255 in the `FreeRTOSconfig.h` file for the demo you are running:

```
#define configLOGGING_MAX_MESSAGE_LENGTH    255
```

Installing a terminal emulator

A terminal emulator can help you diagnose problems or verify that your device code is running properly. There are a variety of terminal emulators available for Windows, macOS, and Linux.

You must connect your board to your computer before you attempt to establish a serial connection to your board with a terminal emulator.

Use the following settings to configure your terminal emulator:

Terminal Setting	Value
BAUD rate	115200
Data	8 bit
Parity	none
Stop	1 bit

Terminal Setting	Value
Flow control	none

Finding your board's serial port

If you do not know your board's serial port, you can issue one of the following commands from the command line or terminal to return the serial ports for all devices connected to your host computer:

Windows

```
chgport
```

Linux

```
ls /dev/tty*
```

macOS

```
ls /dev/cu.*
```

Using CMake with FreeRTOS

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

You can use CMake to generate project build files from FreeRTOS application source code, and to build and run the source code.

You can also use an IDE to edit, debug, compile, flash, and run code on FreeRTOS-qualified devices. Each board-specific Getting Started guide includes instructions for setting up the IDE for

a particular platform. If you prefer working without an IDE, you can use other third-party code editing and debugging tools for developing and debugging your code, and then use CMake to build and run the applications.

The following boards support CMake:

- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- Infineon XMC4800 IoT Connectivity Kit
- Marvell MW320 AWS IoT Starter Kit
- Marvell MW322 AWS IoT Starter Kit
- Microchip Curiosity PIC32MZEF Bundle
- Nordic nRF52840 DK Development kit
- STMicroelectronicsSTM32L4 Discovery Kit IoT Node
- Texas Instruments CC3220SF-LAUNCHXL
- Microsoft Windows Simulator

See the topics below for more information about using CMake with FreeRTOS.

Topics

- Prerequisites
- Developing FreeRTOS applications with third-party code editors and debugging tools
- Building FreeRTOS with CMake

Prerequisites

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

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

  Windows subsystem for Linux (WSL) is not supported. Use native CMake on Windows machines.
- You must have CMake version 3.13 or higher installed.

  You can download the binary distribution of CMake from CMake.org.

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

You can also download and install CMake using a package manager, like homebrew on macOS, and scoop or chocolatey on Windows.

> **Note**
>
> The CMake package versions provided in the package managers for many Linux distributions are out-of-date. If your distribution's package manager does not provide the latest version of CMake, you can try alternative package managers, like `linuxbrew` or `nix`.

- You must have a compatible native build system.

  CMake can target many native build systems, including GNU Make or Ninja. Both Make and Ninja can be installed with package managers on Linux, macOS and Windows. If you are using Make on Windows, you can install a standalone version from Equation, or you can install MinGW, which bundles make.

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

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

Developing FreeRTOS applications with third-party code editors and debugging tools

You can use a code editor and a debugging extension or a third-party debugging tool to develop applications for FreeRTOS.

If, for example, you use Visual Studio Code as your code editor, you can install the Cortex-Debug VS Code extension as a debugger. When you finish developing your application, you can invoke the CMake command-line tool to build your project from within VS Code. For more information about using CMake to build FreeRTOS applications, see Building FreeRTOS with CMake.

For debugging, you can provide a VS Code with debug configuration similar to the following:

```
"configurations": [
    {
        "name": "Cortex Debug",
        "cwd": "${workspaceRoot}",
        "executable": "./build/st/stm32l475_discovery/aws_demos.elf",
        "request": "launch",
        "type": "cortex-debug",
        "servertype": "stutil"
    }
]
```

Building FreeRTOS with CMake

CMake targets your host operating system as the target system by default. To use it for cross compiling, CMake requires a toolchain file, which specifies the compiler that you want to use. In FreeRTOS, we provide default toolchain files in *freertos*/tools/cmake/toolchains. The way to provide this file to CMake depends on whether you're using the CMake command line interface or GUI. For more details, follow the Generating build files (CMake command-line tool) instructions below. For more information about cross-compiling in CMake, see CrossCompiling in the official CMake wiki.

To build a CMake-based project

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

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

   For information about generating FreeRTOS build files, see Generating build files (CMake command-line tool) and Generating build files (CMake GUI).
2. Invoke the native build system to make the project into an executable.

   For information about making FreeRTOS build files, see Building FreeRTOS from generated build files.

Generating build files (CMake command-line tool)

You can use the CMake command-line tool (cmake) to generate build files for FreeRTOS. To generate the build files, you need to specify a target board, a compiler, and the location of the source code and build directory.

You can use the following options for cmake:

- `-DVENDOR` – Specifies the target board.
- `-DCOMPILER` – Specifies the compiler.
- `-S` – Specifies the location of the source code.
- `-B` – Specifies the location of generated build files.

> **Note**
>
> The compiler must be in the system's PATH variable, or you must specify the location of the compiler.

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

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

> **Note**
>
> If you are using Windows, you must specify the native build system because CMake uses Visual Studio by default. For example:
>
> ```
> cmake -DVENDOR=ti -DBOARD=cc3220_launchpad -DCOMPILER=arm-ti -S . -B build-
> directory -G Ninja
> ```
>
> Or:

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

The regular expressions `${VENDOR}.*` and `${BOARD}.*` are used to search for a matching board, so you don't have to use the full names of the vendor and board for the `VENDOR` and `BOARD` options. Partial names work, provided there is a single match. For example, the following commands generate the same build files from the same source:

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

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

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

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

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

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

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

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

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

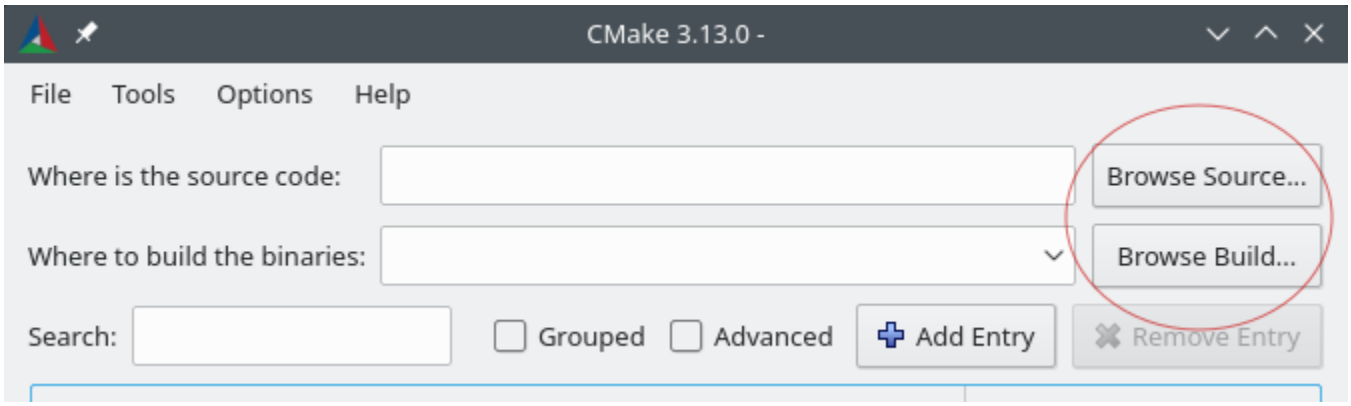
You can also set the `CMAKE_BUILD_TYPE` to `release` to add optimization flags to the compile options.

Generating build files (CMake GUI)

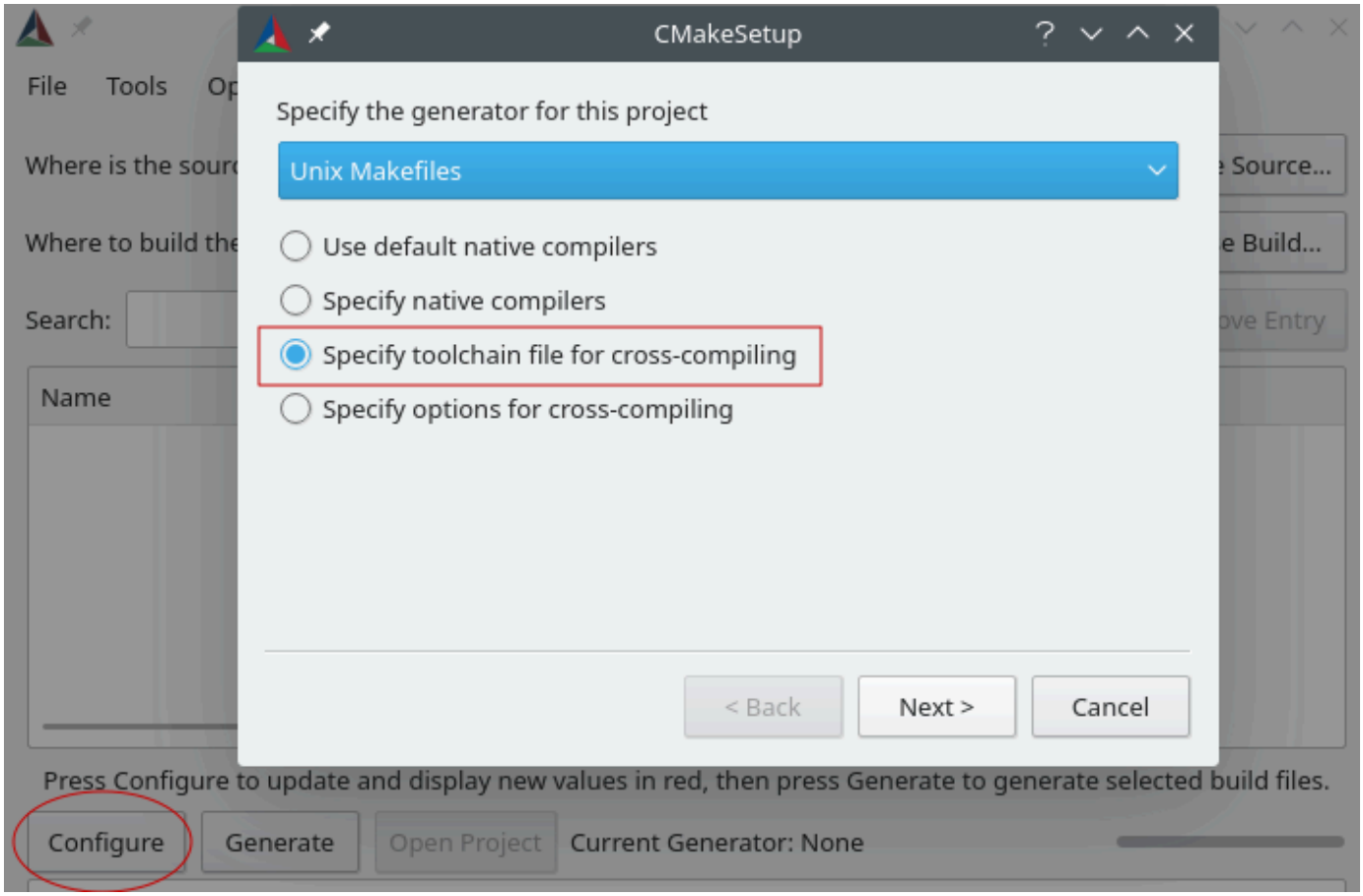
You can use the CMake GUI to generate FreeRTOS build files.

To generate build files with the CMake GUI

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



3. Choose **Configure**, and under **Specify the build generator for this project**, find and choose the build system that you want to use to build the generated build files. if you do not see the pop up window, you might be reusing an existing build directory. In this case, delete the CMake cache by choosing **Delete Cache** from the **File** menu.



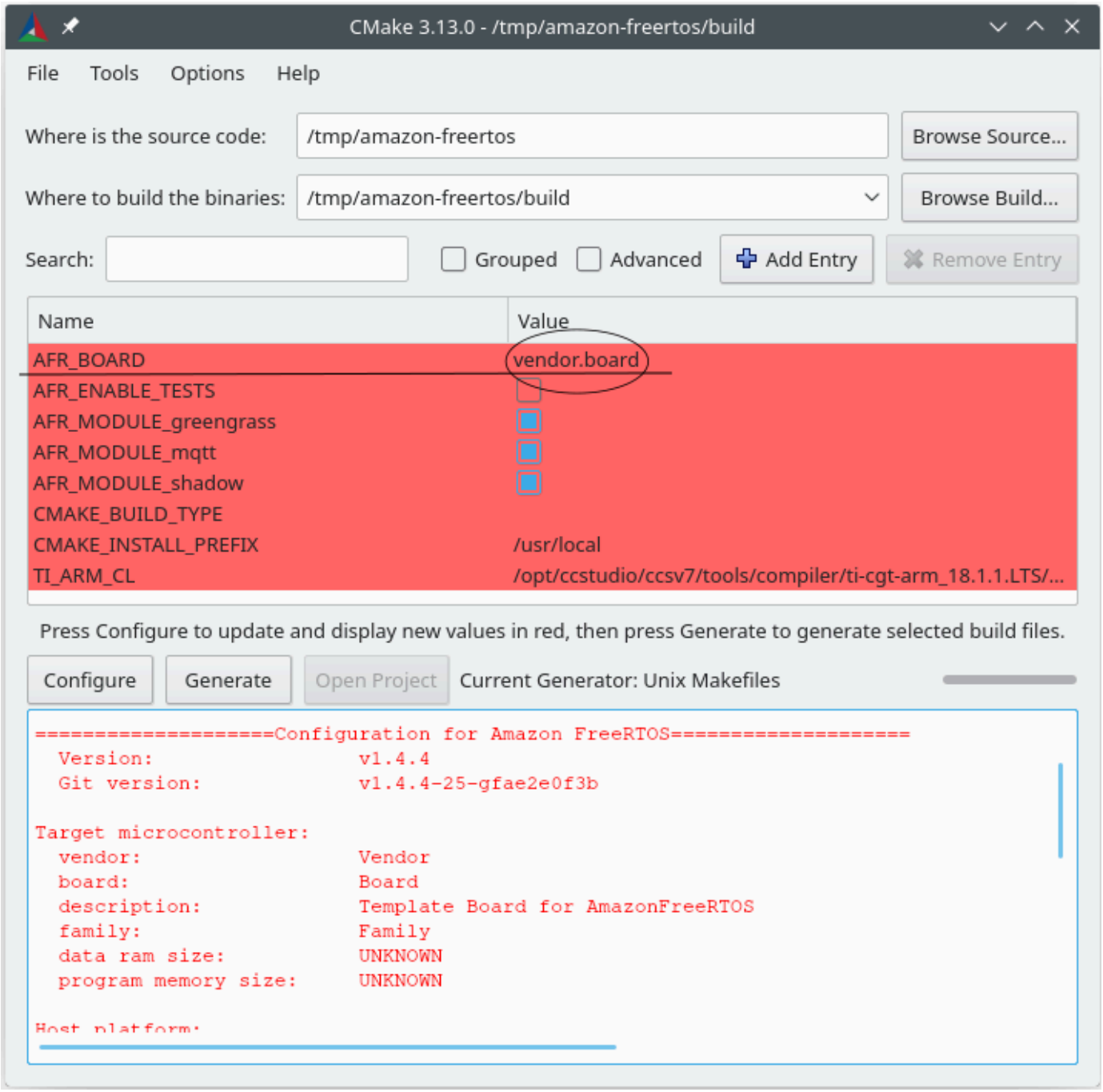
4. Choose **Specify toolchain file for cross-compiling**, and then choose **Next**.
5. Choose the toolchain file (for example, *freertos*`/tools/cmake/toolchains/arm-ti.cmake`), and then choose **Finish**.

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

> **Note**
>
> If you are seeing the following error:
>
> ```
> CMake Error at tools/cmake/toolchains/find_compiler.cmake:23 (message):
> Compiler not found, you can specify search path with AFR_TOOLCHAIN_PATH.
> ```

It means the compiler is not in your PATH environment variable. You can set the AFR_TOOLCHAIN_PATH variable in the GUI to tell CMake where you installed your compiler. If you do not see the AFR_TOOLCHAIN_PATH variable, choose **Add Entry**. In the pop up window, under **Name**, type **AFR_TOOLCHAIN_PATH**. Under **Compiler Path** type the path to your compiler. for example, `C:/toolchains/arm-none-eabi-gcc`.

6. The GUI should now look like this:



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

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

Building FreeRTOS from generated build files

Building with native build system

You can build FreeRTOS with a native build system by calling the build system command from the output binaries directory.

For example, if your build file output directory is `<build_dir>`, and you are using Make as your native build system, run the following commands:

```
cd <build_dir>
make -j4
```

Building with CMake

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

```
cmake --build build_dir
```

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

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

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

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

For more information about the CMake build mode, see the CMake documentation.

Developer-mode key provisioning

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

This section discusses two options to get a trusted X.509 client certificate onto an IoT device for lab testing. Depending on the capabilities of the device, various provisioning-related operations may or may not be supported, including onboard ECDSA key generation, private key import, and X.509 certificate enrollment. In addition, different use cases call for different levels of key protection, ranging from onboard flash storage to the use of dedicated crypto hardware. This section provides logic for working within the cryptographic capabilities of your device.

Option #1: private key import from AWS IoT

For lab testing purposes, if your device allows the import of private keys, follow the instructions in Configuring the FreeRTOS demos.

Option #2: onboard private key generation

If your device has a secure element, or if you prefer to generate your own device key pair and certificate, follow the instructions here.

Initial Configuration

First, perform the steps in Configuring the FreeRTOS demos, but skip the last step (that is, don't do *To format your AWS IoT credentials*). The net result should be that the `demos/include/aws_clientcredential.h` file has been updated with your settings, but the `demos/include/aws_clientcredential_keys.h` file has not.

Demo Project Configuration

Open the coreMQTT Mutual Authentication demo as described in the guide for your board in Board-specific getting started guides . In the project, open

the file `aws_dev_mode_key_provisioning.c` and change the definition of `keyprovisioningFORCE_GENERATE_NEW_KEY_PAIR`, which is set to zero by default, to one:

```
#define keyprovisioningFORCE_GENERATE_NEW_KEY_PAIR 1
```

Then build and run the demo project and continue to the next step.

Public Key Extraction

Because the device hasn't been provisioned with a private key and client certificate, the demo will fail to authenticate to AWS IoT. However, the coreMQTT Mutual Authentication demo starts by running developer-mode key provisioning, resulting in the creation of a private key if one was not already present. You should see something like the following near the beginning of the serial console output.

```
7 910 [IP-task] Device public key, 91 bytes:
3059 3013 0607 2a86 48ce 3d02 0106 082a
8648 ce3d 0301 0703 4200 04cd 6569 ceb8
1bb9 1e72 339f e8cf 60ef 0f9f b473 33ac
6f19 1813 6999 3fa0 c293 5fae 08f1 1ad0
41b7 345c e746 1046 228e 5a5f d787 d571
dcb2 4e8d 75b3 2586 e2cc 0c
```

Copy the six lines of key bytes into a file called `DevicePublicKeyAsciiHex.txt`. Then use the command-line tool "xxd" to parse the hex bytes into binary:

```
xxd -r -ps DevicePublicKeyAsciiHex.txt DevicePublicKeyDer.bin
```

Use "openssl" to format the binary encoded (DER) device public key as PEM:

```
openssl ec -inform der -in DevicePublicKeyDer.bin -pubin -pubout -outform pem -out
 DevicePublicKey.pem
```

Don't forget to disable the temporary key generation setting you enabled above. Otherwise, the device will create yet another key pair, and you will have to repeat the previous steps:

```
#define keyprovisioningFORCE_GENERATE_NEW_KEY_PAIR 0
```

Public Key Infrastructure Setup

Follow the instructions in Registering Your CA Certificate to create a certificate hierarchy for your device lab certificate. Stop before executing the sequence described in the section *Creating a Device Certificate Using Your CA Certificate*.

In this case, the device will not be signing the certificate request (that is, the Certificate Service Request or CSR) because the X.509 encoding logic required for creating and signing a CSR has been excluded from the FreeRTOS demo projects to reduce ROM size. Instead, for lab testing purposes, create a private key on your workstation and use it to sign the CSR.

```
openssl genrsa -out tempCsrSigner.key 2048
openssl req -new -key tempCsrSigner.key -out deviceCert.csr
```

Once your Certificate Authority has been created and registered with AWS IoT, use the following command to issue a client certificate based on the device CSR that was signed in the previous step:

```
openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key
 -CAcreateserial -out deviceCert.pem -days 500 -sha256 -force_pubkey
 DevicePublicKey.pem
```

Even though the CSR was signed with a temporary private key, the issued certificate can only be used with the actual device private key. The same mechanism can be used in production if you store the CSR signer key in separate hardware, and configure your certificate authority so that it only issues certificates for requests that have been signed by that specific key. That key should also remain under the control of a designated administrator.

Certificate Import

With the certificate issued, the next step is to import it into your device. You will also need to import your Certificate Authority (CA) certificate, since it is required in order for first-time authentication to AWS IoT to succeed when using JITP. In the `aws_clientcredential_keys.h` file in your project, set the `keyCLIENT_CERTIFICATE_PEM` macro to be the contents of deviceCert.pem and set the `keyJITR_DEVICE_CERTIFICATE_AUTHORITY_PEM` macro to be the contents of `rootCA.pem`.

Device Authorization

Import `deviceCert.pem` into the AWS IoT registry as described in Use Your Own Certificate. You must create a new AWS IoT thing, attach the PENDING certificate and a policy to your

thing, then mark the certificate as ACTIVE. All of these steps can be performed manually in the AWS IoT console.

Once the new client certificate is ACTIVE and associated with a thing and a policy, run the coreMQTT Mutual Authentication demo again. This time, the connection to the AWS IoT MQTT broker will succeed.

Board-specific getting started guides

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

After you complete the First steps, see your board's guide for board-specific instructions on getting started with FreeRTOS:

- Getting started with the Cypress CYW943907AEVAL1F Development Kit
- Getting started with the Cypress CYW954907AEVAL1F Development Kit
- Getting started with the Cypress CY8CKIT-064S0S2-4343W kit
- Getting started with the Infineon XMC4800 IoT Connectivity Kit
- Getting started with the MW32x AWS IoT Starter Kit
- Getting started with the MediaTek MT7697Hx development kit
- Getting started with the Microchip Curiosity PIC32MZ EF
- Getting started with the Nuvoton NuMaker-IoT-M487
- Getting started with the NXP LPC54018 IoT Module
- Getting started with the Renesas Starter Kit+ for RX65N-2MB
- Getting started with the STMicroelectronics STM32L4 Discovery Kit IoT Node
- Getting started with the Texas Instruments CC3220SF-LAUNCHXL
- Getting started with the Windows Device Simulator
- Getting started with the Xilinx Avnet MicroZed Industrial IoT Kit

Note

You do not need to complete the First steps for the following self-contained Getting Started with FreeRTOS guides:

- Getting started with the Microchip ATECC608A Secure Element with Windows simulator
- Getting started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT
- Getting started with the Espressif ESP32-WROOM-32SE
- Getting started with the Espressif ESP32-S2
- Getting started with the Infineon OPTIGA Trust X and XMC4800 IoT Connectivity Kit
- Getting started with the Nordic nRF52840-DK

Getting started with the Cypress CYW943907AEVAL1F Development Kit

Important

This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Cypress CYW943907AEVAL1F Development Kit. If you do not have the Cypress CYW943907AEVAL1F Development Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

Note

This tutorial walks you through setting up and running the coreMQTT Mutual Authentication demo. The FreeRTOS port for this board currently does not support the TCP server and client demos.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions.

Important

- In this topic, the path to the FreeRTOS download directory is referred to as *`freertos`*.
- Space characters in the *`freertos`* path can cause build failures. When you clone or copy the repository, make sure the path that you create doesn't contain space characters.
- The maximum length of a file path on Microsoft Windows is 260 characters. Long FreeRTOS download directory paths can cause build failures.
- Because the source code may contain symbolic links, if you're using Windows to extract the archive, you may have to:
  - Enable Developer Mode or,
  - Use a console that is elevated as administrator.

  In this way, Windows can properly create symbolic links when it extracts the archive. Otherwise, symbolic links will be written as normal files that contain the paths of the symbolic links as text or are empty. For more information, see the blog entry Symlinks in Windows 10!.

  If you use Git under Windows, you must enable Developer Mode or you must:
  - Set `core.symlinks` to true with the following command:

    ```
    git config --global core.symlinks true
    ```
  - Use a console that is elevated as administrator whenever you use a git command that writes to the system (for example, **git pull**, **git clone**, and **git submodule update --init --recursive**).
- As noted in Downloading FreeRTOS, FreeRTOS ports for Cypress are currently only available on GitHub.

Overview

This tutorial contains instructions for the following getting started steps:

1. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
2. Cross compiling a FreeRTOS demo application to a binary image.

3. Loading the application binary image to your board, and then running the application.
4. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Setting up your development environment

Download and install the WICED Studio SDK

In this Getting Started guide, you use the Cypress WICED Studio SDK to program your board with the FreeRTOS demo. Visit the WICED Software website to download the WICED Studio SDK from Cypress. You must register for a free Cypress account to download the software. The WICED Studio SDK is compatible with Windows, macOS, and Linux operating systems.

> **Note**
>
> Some operating systems require additional installation steps. Make sure that you read and follow all installation instructions for the operating system and version of WICED Studio that you are installing.

Set environment variables

Before you use WICED Studio to program your board, you must create an environment variable for the WICED Studio SDK installation directory. If WICED Studio is running while you create your variables, you need to restart the application after you set your variables.

> **Note**
>
> The WICED Studio installer creates two separate folders named `WICED-Studio-`*`m`*`.`*`n`* on your machine where `m` and `n` are the major and minor version numbers respectively. This document assumes a folder name of `WICED-Studio-6.2` but be sure to use the correct name for the version that you install. When you define the `WICED_STUDIO_SDK_PATH` environment variable, be sure to specify the full installation path of the WICED Studio SDK, and not the installation path of the WICED Studio IDE. In Windows and macOS, the `WICED-Studio-`*`m`*`.`*`n`* folder for the SDK is created in the `Documents` folder by default.

To create the environment variable on Windows

1. Open **Control Panel**, choose **System**, and then choose **Advanced System Settings**.
2. On the **Advanced** tab, choose **Environment Variables**.
3. Under **User variables**, choose **New**.
4. For **Variable name**, enter **`WICED_STUDIO_SDK_PATH`**. For **Variable value**, enter the WICED Studio SDK installation directory.

To create the environment variable on Linux or macOS

1. Open the `/etc/profile` file on your machine, and add the following to the last line of the file:

   ```
   export WICED_STUDIO_SDK_PATH=installation-path/WICED-Studio-6.2
   ```

2. Restart your machine.
3. Open a terminal and run the following commands:

   ```
   cd freertos/vendors/cypress/WICED_SDK
   ```

   ```
   perl platform_adjust_make.pl
   ```

   ```
   chmod +x make
   ```

Establishing a serial connection

To establish a serial connection between your host machine and your board

1. Connect the board to your host computer with a USB Standard-A to Micro-B cable.
2. Identify the USB serial port number for the connection to the board on your host computer.
3. Start a serial terminal and open a connection with the following settings:

   - Baud rate: 115200
   - Data: 8 bit
   - Parity: None

- Stop bits: 1
- Flow control: None

For more information about installing a terminal and setting up a serial connection, see Installing a terminal emulator.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

Build and run the FreeRTOS demo project

After you set up a serial connection to your board, you can build the FreeRTOS demo project, flash the demo to your board, and then run the demo.

To build and run the FreeRTOS demo project in WICED Studio

1. Launch WICED Studio.
2. From the **File** menu, choose **Import**. Expand the `General` folder, choose **Existing Projects into Workspace**, and then choose **Next**.
3. In **Select root directory**, select **Browse...**, navigate to the path *`freertos`*`/projects/cypress/CYW943907AEVAL1F/wicedstudio`, and then select **OK**.
4. Under **Projects**, check the box for just the **aws_demo** project. Choose **Finish** to import the project. The target project **aws_demo** should appear in the **Make Target** window.
5. Expand the **WICED Platform** menu and choose **WICED Filters off**.
6. In the **Make Target** window, expand **aws_demo**, right-click the `demo.aws_demo` file, and then choose **Build Target** to build and download the demo to your board. The demo should run automatically after it is built and downloaded to your board.

Troubleshooting

- If you are using Windows, you might receive the following error when you build and run the demo project:

  ```
  : recipe for target 'download_dct' failed
  make.exe[1]: *** [download_dct] Error 1
  ```

  To troubleshoot this error, do the following:

  1. Browse to *WICED-Studio-SDK-PATH*\WICED-Studio-6.2\43xxx_Wi-Fi\tools\OpenOCD\Win32 and double-click on openocd-all-brcm-libftdi.exe.
  2. Browse to *WICED-Studio-SDK-PATH*\WICED-Studio-6.2\43xxx_Wi-Fi\tools\drivers\CYW9WCD1EVAL1 and double-click on InstallDriver.exe.

- If you are using Linux or macOS, you might receive the following error when you build and run the demo project:

  ```
  make[1]: *** [download_dct] Error 127
  ```

  To troubleshoot this error, use the following command to update the libusb-dev package:

  ```
  sudo apt-get install libusb-dev
  ```

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Cypress CYW954907AEVAL1F Development Kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Cypress CYW954907AEVAL1F Development Kit. If you don't have the Cypress CYW954907AEVAL1F Development Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

Note

This tutorial walks you through setting up and running the coreMQTT Mutual Authentication demo. The FreeRTOS port for this board currently doesn't support the TCP server and client demos.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Important

- In this topic, the path to the FreeRTOS download directory is referred to as *`freertos`*.
- Space characters in the *`freertos`* path can cause build failures. When you clone or copy the repository, make sure the path that you create doesn't contain space characters.
- The maximum length of a file path on Microsoft Windows is 260 characters. Long FreeRTOS download directory paths can cause build failures.
- Because the source code may contain symbolic links, if you're using Windows to extract the archive, you may have to:
  - Enable Developer Mode or,
  - Use a console that is elevated as administrator.

  In this way, Windows can properly create symbolic links when it extracts the archive. Otherwise, symbolic links will be written as normal files that contain the paths of the symbolic links as text or are empty. For more information, see the blog entry Symlinks in Windows 10!.

  If you use Git under Windows, you must enable Developer Mode or you must:

  - Set `core.symlinks` to true with the following command:

    ```
    git config --global core.symlinks true
    ```

- Use a console that is elevated as administrator whenever you use a git command that writes to the system (for example, **git pull**, **git clone**, and **git submodule update --init --recursive**).
- As noted in Downloading FreeRTOS, FreeRTOS ports for Cypress are currently only available on GitHub.

Overview

This tutorial contains instructions for the following getting started steps:

1. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
2. Cross compiling a FreeRTOS demo application to a binary image.
3. Loading the application binary image to your board, and then running the application.
4. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Setting up your development environment

Download and install the WICED Studio SDK

In this Getting Started guide, you use the Cypress WICED Studio SDK to program your board with the FreeRTOS demo. Visit the WICED Software website to download the WICED Studio SDK from Cypress. You must register for a free Cypress account to download the software. The WICED Studio SDK is compatible with Windows, macOS, and Linux operating systems.

Note

Some operating systems require additional installation steps. Make sure that you read and follow all installation instructions for the operating system and version of WICED Studio that you are installing.

Set environment variables

Before you use WICED Studio to program your board, you must create an environment variable for the WICED Studio SDK installation directory. If WICED Studio is running while you create your variables, you need to restart the application after you set your variables.

> **Note**
>
> The WICED Studio installer creates two separate folders named `WICED-Studio-m.n` on your machine where `m` and `n` are the major and minor version numbers respectively. This document assumes a folder name of `WICED-Studio-6.2` but be sure to use the correct name for the version that you install. When you define the `WICED_STUDIO_SDK_PATH` environment variable, be sure to specify the full installation path of the WICED Studio SDK, and not the installation path of the WICED Studio IDE. In Windows and macOS, the `WICED-Studio-m.n` folder for the SDK is created in the `Documents` folder by default.

To create the environment variable on Windows

1. Open **Control Panel**, choose **System**, and then choose **Advanced System Settings**.
2. On the **Advanced** tab, choose **Environment Variables**.
3. Under **User variables**, choose **New**.
4. For **Variable name**, enter **`WICED_STUDIO_SDK_PATH`**. For **Variable value**, enter the WICED Studio SDK installation directory.

To create the environment variable on Linux or macOS

1. Open the `/etc/profile` file on your machine, and add the following to the last line of the file:

   ```
   export WICED_STUDIO_SDK_PATH=installation-path/WICED-Studio-6.2
   ```

2. Restart your machine.
3. Open a terminal and run the following commands:

   ```
   cd freertos/vendors/cypress/WICED_SDK
   ```

```
perl platform_adjust_make.pl
```

```
chmod +x make
```

Establishing a serial connection

To establish a serial connection between your host machine and your board

1. Connect the board to your host computer with a USB Standard-A to Micro-B cable.
2. Identify the USB serial port number for the connection to the board on your host computer.
3. Start a serial terminal and open a connection with the following settings:
   - Baud rate: 115200
   - Data: 8 bit
   - Parity: None
   - Stop bits: 1
   - Flow control: None

For more information about installing a terminal and setting up a serial connection, see Installing a terminal emulator.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*****/example/topic**, and then choose **Subscribe to topic**.

Build and run the FreeRTOS demo project

After you set up a serial connection to your board, you can build the FreeRTOS demo project, flash the demo to your board, and then run the demo.

To build and run the FreeRTOS demo project in WICED Studio

1. Launch WICED Studio.
2. From the **File** menu, choose **Import**. Expand the `General` folder, choose **Existing Projects into Workspace**, and then choose **Next**.
3. In **Select root directory**, select **Browse...**, navigate to the path *`freertos`*`/projects/cypress/CYW954907AEVAL1F/wicedstudio`, and then select **OK**.
4. Under **Projects**, check the box for just the **aws_demo** project. Choose **Finish** to import the project. The target project **aws_demo** should appear in the **Make Target** window.
5. Expand the **WICED Platform** menu and choose **WICED Filters off**.
6. In the **Make Target** window, expand **aws_demo**, right-click the `demo.aws_demo` file, and then choose **Build Target** to build and download the demo to your board. The demo should run automatically after it is built and downloaded to your board.

Troubleshooting

- If you are using Windows, you might receive the following error when you build and run the demo project:

  ```
  : recipe for target 'download_dct' failed
  make.exe[1]: *** [download_dct] Error 1
  ```

  To troubleshoot this error, do the following:

  1. Browse to *`WICED-Studio-SDK-PATH`*`\WICED-Studio-6.2\43xxx_Wi-Fi\tools\OpenOCD\Win32` and double-click on `openocd-all-brcm-libftdi.exe`.
  2. Browse to *`WICED-Studio-SDK-PATH`*`\WICED-Studio-6.2\43xxx_Wi-Fi\tools\drivers\CYW9WCD1EVAL1` and double-click on `InstallDriver.exe`.
- If you are using Linux or macOS, you might receive the following error when you build and run the demo project:

```
make[1]: *** [download_dct] Error 127
```

To troubleshoot this error, use the following command to update the libusb-dev package:

```
sudo apt-get install libusb-dev
```

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Cypress CY8CKIT-064S0S2-4343W kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the CY8CKIT-064S0S2-4343W kit. If you don't already have one, you can use that link to purchase a kit. You can also use that link to access the kit user guide.

Getting started

Before you begin, you must configure AWS IoT and FreeRTOS to connect your device to the AWS Cloud. For instructions, see First steps. After you complete the prerequisites, you will have a FreeRTOS package with AWS IoT Core credentials.

> **Note**
>
> In this tutorial, the path to the FreeRTOS download directory created in the "First steps" section is referred to as *`freertos`*.

Setting up the development environment

FreeRTOS works with either a CMake or Make build flow. You can use ModusToolbox for your Make build flow. You can use the Eclipse IDE delivered with ModusToolbox or a partner IDE such as IAR EW-Arm, Arm MDK, or Microsoft Visual Studio Code. The Eclipse IDE is compatible with the Windows, macOS, and Linux operating systems.

Before you begin, download and install the latest ModusToolbox software. For more information, see the ModusToolbox Installation Guide.

Updating tools for ModusToolbox 2.1 or older

If you're using the ModusToolbox 2.1 Eclipse IDE to program this kit, you'll need to update the OpenOCD and Firmware-loader tools.

In the following steps, by default the *ModusToolbox* path for:

- Windows is `C:\Users\`*user_name*`\ModusToolbox`.
- Linux is *user_home*`/ModusToolbox` or where you choose to extract the archive file.
- MacOS is under the Applications folder in the volume you select in the wizard.

Updating OpenOCD

This kit requires Cypress OpenOCD 4.0.0 or later to successfully erase and program the chip.

To update Cypress OpenOCD

1. Go to the Cypress OpenOCD release page.
2. Download the archive file for your OS (Windows/Mac/Linux).
3. Delete the existing files in *ModusToolbox*`/tools_2.x/openocd`.
4. Replace the files in *ModusToolbox*`/tools_2.x/openocd` with the extracted contents of the archive that you downloaded in a previous step.

Updating Firmware-loader

This kit requires Cypress Firmware-loader 3.0.0 or later.

To update Cypress Firmware-loader

1. Go to the Cypress Firmware-loader release page.

2. Download the archive file for your OS (Windows/Mac/Linux).
3. Delete the existing files in *ModusToolbox*`/tools_2.x/fw-loader`.
4. Replace the files in *ModusToolbox*`/tools_2.x/fw-loader` with the extracted contents of the archive that you downloaded in a previous step.

Alternatively, you can use CMake to generate project build files from FreeRTOS application source code, build the project using your preferred build tool, and then program the kit using OpenOCD. If you prefer to use a GUI tool for programming with the CMake flow, download and install Cypress Programmer from the Cypress Programming Solutions webpage. For more information, see Using CMake with FreeRTOS.

Setting up your hardware

Follow these steps to set up the kit's hardware.

1. **Provision your kit**

   Follow the Provisioning Guide for CY8CKIT-064S0S2-4343W Kit instructions to securely provision your kit for AWS IoT.

   This kit requires CySecureTools 3.1.0 or later.

2. **Set up a serial connection**

   a. Connect the kit to your host computer.

   b. The USB Serial port for the kit is automatically enumerated on the host computer. Identify the port number. In Windows, you can identify it using the **Device Manager** under **Ports** (COM & LPT).

   c. Start a serial terminal and open a connection with the following settings:

   - Baud rate: 115200
   - Data: 8 bit
   - Parity: None
   - Stop bits: 1
   - Flow control: None

Build and run the FreeRTOS Demo project

In this section you build and run the demo.

1. Make sure to follow the steps in Provisioning Guide for CY8CKIT-064S0S2-4343W Kit.
2. **Build the FreeRTOS Demo.**

   a. Open the Eclipse IDE for ModusToolbox and choose, or create, a workspace.

   b. From the **File** menu, choose **Import**.

   Expand **General**, choose **Existing Project Into Workspace**, and then choose **Next**.

   c. In **Root Directory**, enter *freertos*`/projects/cypress/CY8CKIT-064S0S2-4343W/mtb/aws_demos` and then select the project name `aws_demos`. It should be selected by default.

   d. Choose **Finish** to import the project into your workspace.

   e. Build the application by doing one of the following:

   - From the **Quick Panel**, select **Build aws_demos Application**.
   - Choose **Project** and choose **Build All**.

   Make sure the project compiles without errors.
3. **Monitoring MQTT Messages on the Cloud**

   Before you run the demo, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud. To subscribe to the MQTT topic with the AWS IoT MQTT client, follow these steps.

   a. Sign in to the AWS IoT console.

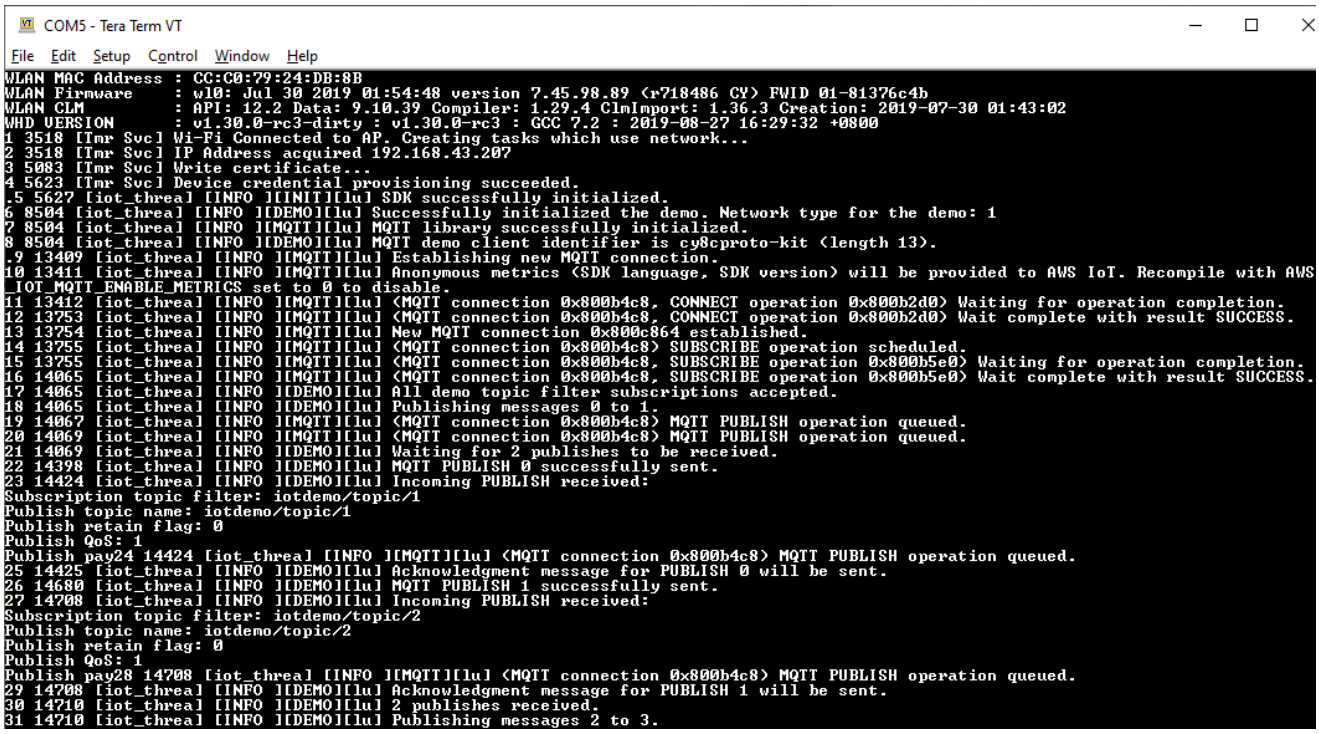
   b. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.

   c. For **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.
4. **Run the FreeRTOS demo project**

   a. Select the project `aws_demos` in the workspace.

   b. From the **Quick Panel**, select **aws_demos Program (KitProg3)**. This programs the board and the demo application starts running after the programming is finished.

c. You can view the status of the running application in the serial terminal. The following figure shows a part of the terminal output.



The MQTT demo publishes messages on four different topics (`iotdemo/topic/`*n*, where n=1 to 4) and subscribes to all those topics to receive the same messages back. When a message is received, the demo publishes an acknowledgement message on the topic `iotdemo/acknowledgements`. The following list describes the debug messages that appear in the terminal output, with references to the serial numbers of the messages. In the output, the WICED Host Driver (WHD) driver details are printed first without serial numbering.

1. 1 to 4 – Device connects to the configured Access Point (AP) and is provisioned by connecting to the AWS server using the configured endpoint and certificates.
2. 5 to 13 – coreMQTT library is initialized and device establishes MQTT connection.
3. 14 to 17 – Device subscribes to all the topics to receive the published messages back.
4. 18 to 30 – Device publishes two messages and waits to receive them back. When each message is received, the device sends an acknowledgement message.

The same cycle of publish, receive, and acknowledge continues until all the messages are published. Two messages are published per cycle until the number of cycles configured are completed.

5. **Using CMake with FreeRTOS**

   You can also use CMake to build and run the demo application. To set up CMake and a native build system, see Prerequisites.

   a. Use the following command to generate build files. Specify the target board with the `-DBOARD` option.

   ```
   cmake -DVENDOR=cypress -DBOARD=CY8CKIT_064S0S2_4343W -DCOMPILER=arm-gcc -
   S freertos -B build_dir
   ```

   If you're using Windows, you must specify the native build system using the `-G` option because CMake uses Visual Studio by default.

   Example

   ```
   cmake -DVENDOR=cypress -DBOARD=CY8CKIT_064S0S2_4343W -DCOMPILER=arm-gcc -
   S freertos -B build_dir -G Ninja
   ```

   If `arm-none-eabi-gcc` is not in your shell path, you also need to set the `AFR_TOOLCHAIN_PATH` CMake variable.

   Example

   ```
   -DAFR_TOOLCHAIN_PATH=/home/user/opt/gcc-arm-none-eabi/bin
   ```

   b. Use the following command to build the project using CMake.

   ```
   cmake --build build_dir
   ```

   c. Finally, program the `cm0.hex` and `cm4.hex` files generated under *build_dir* by using Cypress Programmer.

Running other demos

The following demo applications have been tested and verified to work with the current release. You can find these demos under the *freertos*/demos directory. For information on how to run these demos, see FreeRTOS demos.

- Bluetooth Low Energy demo
- Over-the-Air Updates demo
- Secure Sockets Echo Client demo
- AWS IoT Device Shadow demo

Debugging

The KitProg3 on the kit supports debugging over the SWD protocol.

- To debug the FreeRTOS application, select the **aws_demos project** in the workspace and then select **aws_demos Debug (KitProg3)** from the **Quick Panel**.

OTA updates

PSoC 64 MCUs have passed all of the required FreeRTOS qualification tests. However, the optional over-the-air (OTA) feature implemented in the PSoC 64 Standard Secure AWS firmware library is still pending evaluation. The OTA feature as-implemented currently passes all of the OTA qualification tests except aws_ota_test_case_rollback_if_unable_to_connect_after_update.py.

When a successfully validated OTA image is applied to a device using the PSoC64 Standard Secure – AWS MCU and the device can't communicate with AWS IoT Core, the device can't automatically rollback to the original known good image. This might result in the device being unreachable from AWS IoT Core for further updates. This functionality is still under development by the Cypress team.

For more information, see OTA Updates with AWS and the CY8CKIT-064S0S2-4343W Kit. If you have further questions or need technical support, contact the Cypress Developer Community.

Getting started with the Microchip ATECC608A Secure Element with Windows simulator

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Microchip ATECC608A Secure Element with Windows Simulator.

You need the following hardware:

- Microchip ATECC608A secure element clickboard
- SAMD21 XPlained Pro
- mikroBUS Xplained Pro adapter

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains the following steps:

1. Connect your board to a host machine.
2. Install software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross-compile an FreeRTOS demo application to a binary image.
4. Load the application binary image to your board, and then run the application.

Set up the Microchip ATECC608A hardware

Before you can interact with your Microchip ATECC608A device, you must first program the SAMD21.

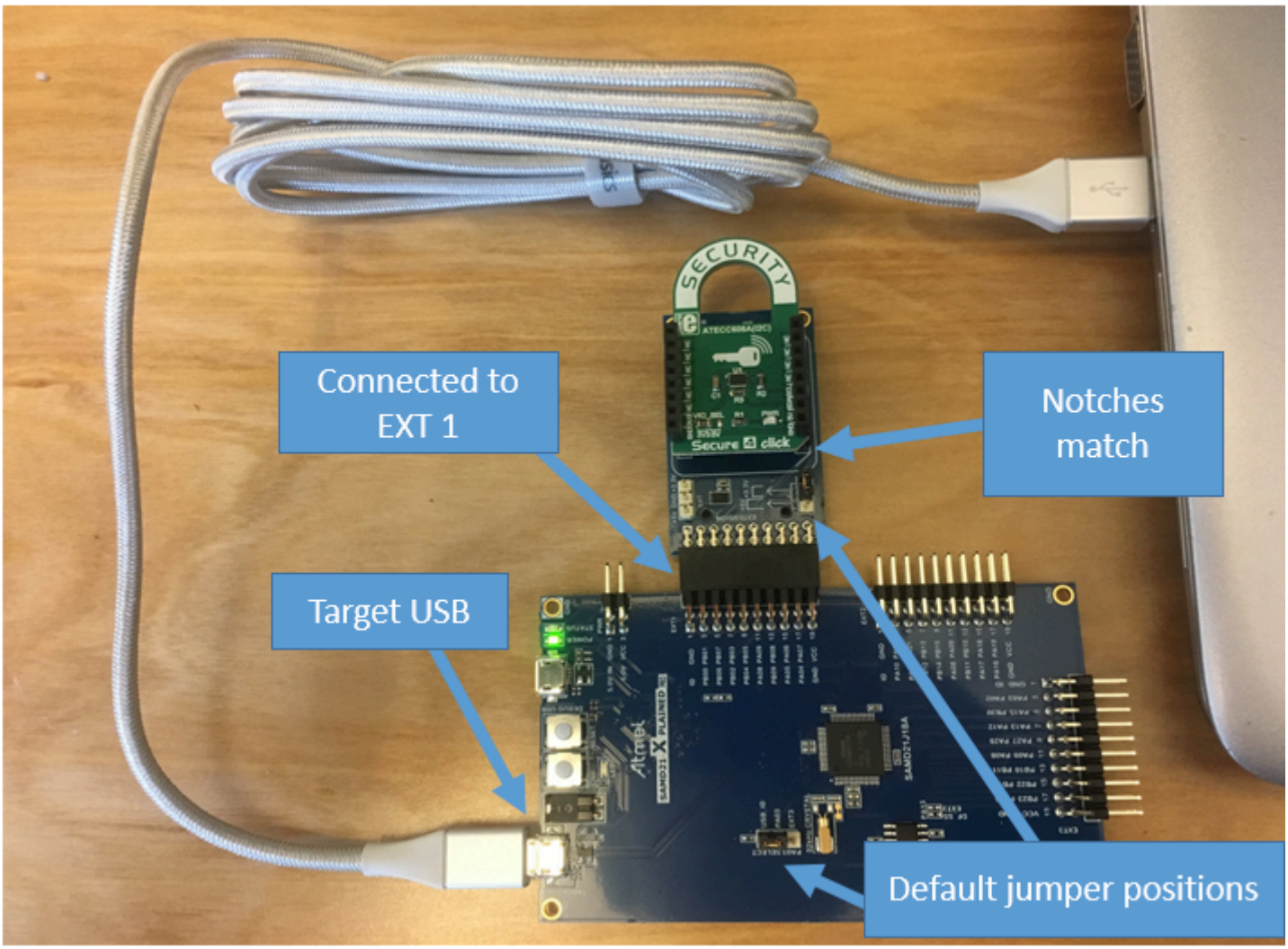
To set up the SAMD21 XPlained Pro board

1. Follow the CryptoAuthSSH-XSTK (DM320109) - Latest Firmware link to download a .zip file containing instructions (PDF) and a binary which can be programmed onto the D21.
2. Download and install the Atmel Studio 7 IDP. Make sure that you select the **SMART ARM MCU** driver architecture during installation.
3. Use a USB 2.0 Micro B cable to attach the "Debug USB" connector to your computer, and follow the instructions in the PDF. (The "Debug USB" connector is the USB port closest to the POWER led and pins.)

To connect the hardware

1. Unplug the micro USB cable from Debug USB.
2. Plug the mikroBUS XPlained Pro adapter into the SAMD21 board in the EXT1 location.
3. Plug the ATECC608A Secure 4 Click board into the mikroBUSX XPlained Pro adapter. Make sure that the notched corner of the click board matches with the notched icon on the adapter board.
4. Plug the micro USB cable into Target USB.

Your setup should look like the following.



Set up your development environment

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign

administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Setting up

1. Download the FreeRTOS repo from the FreeRTOS GitHub repository.

To download FreeRTOS from GitHub:

1. Browse to the FreeRTOS GitHub repository.
2. Choose **Clone or download**.
3. From the command line on your computer, clone the repository to a directory on your host machine.

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

Important

- In this topic, the path to the FreeRTOS download directory is referred to as *`freertos`*.
- Space characters in the *`freertos`* path can cause build failures. When you clone or copy the repository, make sure the path that you create doesn't contain space characters.
- The maximum length of a file path on Microsoft Windows is 260 characters. Long FreeRTOS download directory paths can cause build failures.
- Because the source code may contain symbolic links, if you're using Windows to extract the archive, you may have to:
  - Enable Developer Mode or,
  - Use a console that is elevated as administrator.

  In this way, Windows can properly create symbolic links when it extracts the archive. Otherwise, symbolic links will be written as normal files that contain the paths of the symbolic links as text or are empty. For more information, see the blog entry Symlinks in Windows 10!.

  If you use Git under Windows, you must enable Developer Mode or you must:

  - Set `core.symlinks` to true with the following command:

    ```
    git config -\-global core.symlinks true
    ```

- Use a console that is elevated as administrator whenever you use a git command that writes to the system (for example, **git pull**, **git clone**, and **git submodule update -\-init -\-recursive**).

4. From the *freertos* directory, check out the branch to use.

2. Set up your development environment.

   a. Install the latest version of WinPCap.

   b. Install Microsoft Visual Studio.

      Visual Studio versions 2017 and 2019 are known to work. All editions of these Visual Studio versions are supported (Community, Professional, or Enterprise).

      In addition to the IDE, install the Desktop development with C++ component. Then, under **Optional**, install the latest Windows 10 SDK.

   c. Make sure that you have an active hard-wired Ethernet connection.

Build and run the FreeRTOS demo project

> **Important**
>
> The Microchip ATECC608A device has a one time initialization that is locked onto the device the first time a project is run (during the call to `C_InitToken`). However, the FreeRTOS demo project and test project have different configurations. If the device is locked during the demo project configurations, it will not be possible for all tests in the test project to succeed.

To build and run the FreeRTOS demo project with the Visual Studio IDE

1. Load the project into Visual Studio.

   From the **File** menu, choose **Open**. Choose **File/Solution**, navigate to the *freertos*`\projects\microchip\ecc608a_plus_winsim\visual_studio\aws_demos\aws_demos.sln` file, and then choose **Open**.

2. Retarget the demo project.

The demo project depends on the Windows SDK, but it does not have a Windows SDK version specified. By default, the IDE might attempt to build the demo with an SDK version not present on your machine. To set the Windows SDK version, right-click **aws_demos**, and then choose **Retarget Projects**. This opens the **Review Solution Actions** window. Choose a Windows SDK version that is present on your machine (use the initial value in the drop-down list), and then choose **OK**.

3. Build and run the project.

   From the **Build** menu, choose **Build Solution**, and make sure the solution builds without errors. Choose **Debug, Start Debugging** to run the project. On the first run, you need to configure your device interface and recompile. For more information, see Configure your network interface.

4. Provision the Microchip ATECC608A.

   Microchip has provided several scripting tools to help with the setup of the ATECC608A parts. Navigate to *freertos*`\vendors\microchip\secure_elements\app\example_trust_chain_tool`, and open the README.md file.

   Follow the instructions in the `README.md` file to provision your device. The steps include the following:

   1. Create and register a certificate authority with AWS.
   2. Generate your keys on the Microchip ATECC608A and export the public key and device serial number.
   3. Generate a certificate for the device and registering that certificate with AWS.
   4. Load the CA certificate and device certificate onto the device.

5. Build and run FreeRTOS samples.

   Re-run the demo project again. This time you should connect!

Troubleshooting

For general troubleshooting information, see Troubleshooting getting started.

Getting started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT

Important

This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Note

To explore how to integrate FreeRTOS modular libraries and demos within your own Espressif IDF project, see our featured reference integration for ESP32-C3 platform.

Follow this tutorial to get started with the Espressif ESP32-DevKitC equipped with ESP32-WROOM-32, ESP32-SOLO-1, or ESP-WROVER modules and the ESP-WROVER-KIT-VB. To purchase one from our partner on the AWS Partner Device catalog, use the following links:

- ESP32-WROOM-32 DevKitC
- ESP32-SOLO-1
- ESP32-WROVER-KIT

These versions of development boards are supported on FreeRTOS.

For more information about the latest versions of these boards, see ESP32-DevKitC V4 or ESP-WROVER-KIT v4.1 on the Espressif website.

Note

Currently, the FreeRTOS port for ESP32-WROVER-KIT and ESP DevKitC doesn't support the Symmetric multiprocessing (SMP) feature.

Overview

This tutorial guides you through the following steps:

1. Connecting your board to a host machine.
2. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross compiling a FreeRTOS demo application to a binary image.
4. Loading the application binary image to your board, and then running the application.
5. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Prerequisites

Before you get started with FreeRTOS on your Espressif board, you must set up your AWS account and permissions.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Get started

> **Note**
>
> The Linux commands in this tutorial require that you use the Bash shell.

1. **Set up the Espressif hardware.**
   - For information about setting up the ESP32-DevKitC development board hardware, see the ESP32-DevKitC V4 Getting Started Guide.
   - For information about setting up the ESP-WROVER-KIT development board hardware, see the ESP-WROVER-KIT V4.1 Getting Started Guide.

> **Important**
>
> When you reach the **Get Started** section of the Espressif guides, stop, and then return to the instructions on this page.

2. Download Amazon FreeRTOS from GitHub. (For instructions, see the README.md file.)
3. **Set up your development environment**.

   To communicate with your board, you must install a toolchain. Espressif provides the ESP-IDF to develop software for their boards. Since the ESP-IDF has its own version of the FreeRTOS Kernel integrated as a component, Amazon FreeRTOS includes a custom version of the ESP-IDF v4.2 that has the FreeRTOS Kernel removed. This fixes problems with duplicate files when you compile. To use the custom version of the ESP-IDF v4.2 included with Amazon FreeRTOS, follow the instructions below for your host machine's operating system.

   Windows

   1. Download ESP-IDF's Universal Online Installer for Windows.
   2. Run the **Universal Online Installer**.
   3. When you get to the step **Download or use ESP-IDF**, select **Use an existing ESP-IDF directory** and set **Choose existing ESP-IDF directory** to *`freertos`*`/vendors/espressif/esp-idf`.
   4. Complete the installation.

   macOS

   1. Follow the instructions in the Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for macOS.

      > **Important**
      >
      > When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop, and then return to the instructions on this page.

   2. Open a command line window.

3. Navigate to the FreeRTOS download directory, and then run the following script to download and install the espressif toolchain for your platform.

```
vendors/espressif/esp-idf/install.sh
```

4. Add the ESP-IDF toolchain tools to your terminal's path with the following command.

```
source vendors/espressif/esp-idf/export.sh
```

Linux

1. Follow the instructions in the Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for Linux.

> **Important**
>
> When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop, and then return to the instructions on this page.

2. Open a command line window.
3. Navigate to the FreeRTOS download directory, and then run the following script to download and install the Espressif toolchain for your platform.

```
vendors/espressif/esp-idf/install.sh
```

4. Add the ESP-IDF toolchain tools to your terminal's path with the following command.

```
source vendors/espressif/esp-idf/export.sh
```

4. **Establish a serial connection.**

a. To establish a serial connection between your host machine and the ESP32-DevKitC, you must install the CP210x USB to UART Bridge VCP drivers. You can download these drivers from Silicon Labs.

To establish a serial connection between your host machine and the ESP32-WROVER-KIT, you must install the FTDI virtual COM port driver. You can download this driver from FTDI.

b. Follow the steps to Establish Serial Connection with ESP32.

c. After you establish a serial connection, make a note of the serial port for your board's connection. You need it to flash the demo.

Configure the FreeRTOS demo applications

For this tutorial, the FreeRTOS configuration file is located at *freertos*/vendors/espressif/boards/*board-name*/aws_demos/config_files/FreeRTOSConfig.h. (For example, if AFR_BOARD espressif.esp32_devkitc is chosen, the configuration file is located at *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/FreeRTOSConfig.h.)

1. If you're running macOS or Linux, open a terminal prompt. If you're running Windows, open the "ESP-IDF 4.x CMD" app (if you included this option when you installed the ESP-IDF toolchain), or the "Command Prompt" app otherwise.
2. To verify that you have Python3 installed, run

   ```
   python --version
   ```

   The version installed is displayed. If you don't have Python 3.0.1 or later installed, you can install it from the Python website.
3. You need the AWS Command Line Interface (CLI) to run AWS IoT commands. If you're running Windows, use the easy_install awscli command to install the AWS CLI in the "Command" or "ESP-IDF 4.x CMD" app.

   If you're running macOS or Linux, see Installing the AWS CLI.
4. Run

   ```
   aws configure
   ```

   and configure the AWS CLI with your AWS access key ID, secret access key, and default AWS Region. For more information, see Configuring the AWS CLI.
5. Use the following command to install the AWS SDK for Python (boto3):

   - On Windows, in the "Command" or "ESP-IDF 4.x CMD" app, run

     ```
     pip install boto3 --user
     ```

> **Note**
>
> See Boto3 documentation for details.

- On macOS or Linux, run

```
pip install tornado nose --user
```

and then run

```
pip install boto3 --user
```

FreeRTOS includes the `SetupAWS.py` script to make it easier to set up your Espressif board to connect to AWS IoT. To configure the script, open *`freertos`*`/tools/aws_config_quick_start/configure.json` and set the following attributes:

`afr_source_dir`

The complete path to the *`freertos`* directory on your computer. Make sure that you use forward slashes to specify this path.

`thing_name`

The name that you want to assign to the AWS IoT thing that represents your board.

`wifi_ssid`

The SSID of your Wi-Fi network.

`wifi_password`

The password for your Wi-Fi network.

`wifi_security`

The security type for your Wi-Fi network.

The following are valid security types:

- `eWiFiSecurityOpen` (Open, no security)
- `eWiFiSecurityWEP` (WEP security)

- `eWiFiSecurityWPA` (WPA security)
- `eWiFiSecurityWPA2` (WPA2 security)

6. Run the configuration script.

   a. If you're running macOS or Linux, open a terminal prompt. If you're running Windows, open the "ESP-IDF 4.x CMD" or "Command" app.

   b. Navigate to the *freertos*`/tools/aws_config_quick_start` directory and run

   ```
   python SetupAWS.py setup
   ```

   The script does the following:

   - Creates an IoT thing, certificate, and policy.
   - Attaches the IoT policy to the certificate and the certificate to the AWS IoT thing.
   - Populates the `aws_clientcredential.h` file with your AWS IoT endpoint, Wi-Fi SSID, and credentials.
   - Formats your certificate and private key and writes them to the `aws_clientcredential_keys.h` header file.

   > **Note**
   >
   > The certificate is hardcoded for demonstration purposes only. Production-level applications should store these files in a secure location.

   For more information about `SetupAWS.py`, see the `README.md` in the *freertos*`/tools/aws_config_quick_start` directory.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Navigate to the AWS IoT console.

2. In the navigation pane, choose **Test**, then choose **MQTT Test Client**.
3. In **Subscription topic**, enter *your-thing-name*/example/topic, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build, flash, and run the FreeRTOS demo project using the idf.py script

You can use Espressif's IDF utility (`idf.py`) to build the project and flash the binaries onto your device.

> **Note**
>
> Some setups might require that you use the port option `"-p port-name"` with `idf.py` to specify the correct port, as in the following example.
>
> ```
> idf.py -p /dev/cu.usbserial-00101301B flash
> ```

Build and flash FreeRTOS on Windows, Linux, and macOS (ESP-IDF v4.2)

1. Navigate to the root of your FreeRTOS download directory.
2. In a command line window, enter the following command to add the ESP-IDF tools to your terminal's PATH.

   Windows ("Command" app)

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Windows ("ESP-IDF 4.x CMD" app)

   (This has already been done when you opened the app.)

   Linux / macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

3. Configure cmake in the `build` directory and build the firmware image with the following command.

```
idf.py -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 build
```

You should see output like the following.

```
   Running cmake in directory /path/to/hello_world/build
   Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
   Warn about uninitialized values.
   -- Found Git: /usr/bin/git (found version "2.17.0")
   -- Building empty aws_iot component due to configuration
   -- Component names: ...
   -- Component paths: ...

   ... (more lines of build system output)

   [527/527] Generating hello-world.bin
   esptool.py v2.3.1

   Project build complete. To flash, run this command:
   ../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600
 write_flash --flash_mode dio --flash_size detect --flash_freq 40m 0x10000 build/
hello-world.bin  build 0x1000 build/bootloader/bootloader.bin 0x8000 build/
partition_table/partition-table.bin
   or run 'idf.py -p PORT flash'
```

If there are no errors, the build will generate the firmware binary .bin files.

4. Erase your development board's flash memory with the following command.

```
idf.py erase_flash
```

5. Use the `idf.py` script to flash the application binary to your board.

```
idf.py flash
```

6. Monitor the output from your board's serial port with the following command.

```
idf.py monitor
```

> **Note**
>
> You can combine these commands such as in the following example.
>
> ```
> idf.py erase_flash flash monitor
> ```

For certain host machine setups, you must specify the port when you flash the board such as in the following example.

```
idf.py erase_flash flash monitor -p /dev/ttyUSB1
```

Build and Flash FreeRTOS with CMake

In addition to the `idf.py` script provided by the IDF SDK to build and run your code, you can also build the project with CMake. Currently, it supports either Unix Makefiles or the Ninja build system.

To build and flash the project

1. In a command line window, navigate to the root of your FreeRTOS download directory.
2. Run the following script to add the ESP-IDF tools to your shell's PATH.

   Windows

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Linux / macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

3. Enter the following command to generate the build files.

   With Unix Makefiles

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 -S . -
   B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0
   ```

With Ninja

```
cmake -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 -S . -
B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0 -GNinja
```

4. Build the project.

With Unix Makefiles

```
make -C ./YOUR_BUILD_DIRECTORY -j8
```

With Ninja

```
ninja -C ./YOUR_BUILD_DIRECTORY -j8
```

5. Erase the flash and then flash the board.

With Unix Makefiles

```
make -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
make -C ./YOUR_BUILD_DIRECTORY flash
```

With Ninja

```
ninja -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
ninja -C ./YOUR_BUILD_DIRECTORY flash
```

Run the Bluetooth Low Energy demos

FreeRTOS supports Bluetooth Low Energy library connectivity.

To run the FreeRTOS demo project across Bluetooth Low Energy, you must run the FreeRTOS Bluetooth Low Energy Mobile SDK Demo Application on an iOS or Android mobile device.

To set up the FreeRTOS Bluetooth Low Energy mobile SDK demo application

1. Follow the instructions in Mobile SDKs for FreeRTOS Bluetooth devices to download and install the SDK for your mobile platform on your host computer.
2. Follow the instructions in FreeRTOS Bluetooth Low Energy Mobile SDK demo application to set up the demo mobile application on your mobile device.

For instructions about how to run the MQTT over Bluetooth Low Energy demo on your board, see MQTT over Bluetooth Low Energy.

For instructions about how to run the Wi-Fi provisioning demo on your board, see Wi-Fi provisioning.

Using FreeRTOS in your own CMake project for ESP32

If you want to consume FreeRTOS in your own CMake project, you can set it up as a subdirectory and build it together with your application. First, get a copy of FreeRTOS from GitHub. You can also set it up as a Git submodule with the following command so it's easier to update in the future.

```
git submodule add -b release https://github.com/aws/amazon-freertos.git freertos
```

If a later version is released, you can update your local copy with these commands.

```
# Pull the latest changes from the remote tracking branch.
git submodule update --remote -- freertos
```

```
# Commit the submodule change because it is pointing to a different revision now.
git add freertos
```

```
git commit -m "Update FreeRTOS to a new release"
```

If your project has the following directory structure:

```
- freertos (the copy that you obtained from GitHub or the AWS IoT console)
- src
  - main.c (your application code)
- CMakeLists.txt
```

Then the following is an example of the top-level `CMakeLists.txt` file that can be used to build your application together with FreeRTOS.

```
cmake_minimum_required(VERSION 3.13)

project(freertos_examples)

# Tell IDF build to link against this target.
set(IDF_EXECUTABLE_SRCS "<complete_path>/src/main.c")
set(IDF_PROJECT_EXECUTABLE my_app)

# Add FreeRTOS as a subdirectory. AFR_BOARD tells which board to target.
set(AFR_BOARD espressif.esp32_devkitc CACHE INTERNAL "")
add_subdirectory(freertos)

# Link against the mqtt library so that we can use it. Dependencies are transitively
# linked.
target_link_libraries(my_app PRIVATE AFR::core_mqtt)
```

To build the project, run the following CMake commands. Make sure the ESP32 compiler is in the PATH environment variable.

```
cmake -S . -B build-directory -DCMAKE_TOOLCHAIN_FILE=freertos/tools/cmake/toolchains/
xtensa-esp32.cmake -GNinja
```

```
cmake --build build-directory
```

To flash the application to your board, run the following command.

```
cmake --build build-directory --target flash
```

Using components from FreeRTOS

After running CMake, you can find all available components in the summary output. It should look something like the following example.

```
====================Configuration for FreeRTOS====================
  Version:                 202107.00
  Git version:             202107.00-g79ad6defb

Target microcontroller:
```

```
  vendor:                   Espressif
  board:                    ESP32-DevKitC
  description:              Development board produced by Espressif that comes in two
                            variants either with ESP-WROOM-32 or ESP32-WROVER module
  family:                   ESP32
  data ram size:            520KB
  program memory size:      4MB

Host platform:
  OS:                       Linux-4.15.0-66-generic
  Toolchain:                xtensa-esp32
  Toolchain path:           /opt/xtensa-esp32-elf
  CMake generator:          Ninja

FreeRTOS modules:
  Modules to build:         backoff_algorithm, common, common_io, core_http,
                            core_http_demo_dependencies, core_json, core_mqtt,
                            core_mqtt_agent, core_mqtt_agent_demo_dependencies,
                            core_mqtt_demo_dependencies, crypto, defender, dev_mode_key_
                            provisioning, device_defender, device_defender_demo_
                            dependencies, device_shadow,
 device_shadow_demo_dependencies,
                            freertos_cli_plus_uart, freertos_plus_cli, greengrass,
                            http_demo_helpers, https, jobs, jobs_demo_dependencies,
                            kernel, logging, mqtt, mqtt_agent_interface, mqtt_demo_
                            helpers, mqtt_subscription_manager, ota, ota_demo_
                            dependencies, ota_demo_version, pkcs11, pkcs11_helpers,
                            pkcs11_implementation, pkcs11_utils, platform,
 secure_sockets,
                            serializer, shadow, tls, transport_interface_secure_sockets,
                            wifi
   Enabled by user:         common_io, core_http_demo_dependencies, core_json,
                            core_mqtt_agent_demo_dependencies, core_mqtt_demo_
                            dependencies, defender, device_defender,
 device_defender_demo_
                            dependencies, device_shadow,
 device_shadow_demo_dependencies,
                            freertos_cli_plus_uart, freertos_plus_cli, greengrass,
 https,
                            jobs, jobs_demo_dependencies, logging,
 ota_demo_dependencies,
                            pkcs11, pkcs11_helpers, pkcs11_implementation, pkcs11_utils,
                            platform, secure_sockets, shadow, wifi
   Enabled by dependency:   backoff_algorithm, common, core_http, core_mqtt,
```

```
                            core_mqtt_agent, crypto, demo_base,
 dev_mode_key_provisioning,
                            freertos, http_demo_helpers, kernel, mqtt, mqtt_agent_
                            interface, mqtt_demo_helpers, mqtt_subscription_manager,
 ota,
                            ota_demo_version, pkcs11_mbedtls, serializer, tls,
                            transport_interface_secure_sockets, utils
  3rdparty dependencies:    jsmn, mbedtls, pkcs11, tinycbor
  Available demos:          demo_cli_uart, demo_core_http, demo_core_mqtt,
 demo_core_mqtt_
                            agent, demo_device_defender, demo_device_shadow,
                            demo_greengrass_connectivity, demo_jobs, demo_ota_core_http,
                            demo_ota_core_mqtt, demo_tcp
  Available tests:
=============================================================================
```

You can reference any components from the `Modules to build` list. To link them into your application, put the `AFR::` namespace in front of the name, for example, `AFR::core_mqtt`, `AFR::ota`, and so on.

Add custom components using ESP-IDF

You can add more components while using ESP-IDF. For example, assuming you want to add a component called `example_component`, and your project looks like this:

```
- freertos
- components
  - example_component
    - include
      - example_component.h
    - src
      - example_component.c
    - CMakeLists.txt
- src
  - main.c
- CMakeLists.txt
```

The following is an example of the `CMakeLists.txt` file for your component.

```
add_library(example_component src/example_component.c)
target_include_directories(example_component PUBLIC include)
```

Then, in the top level `CMakeLists.txt` file, add the component by inserting the following line just after `add_subdirectory(freertos)`.

```
add_subdirectory(component/example_component)
```

Then, modify `target_link_libraries` to include your component.

```
target_link_libraries(my_app PRIVATE AFR::core_mqtt PRIVATE example_component)
```

This component is now automatically linked to your application code by default. You can now include its header files and call the functions it defines.

Override the configurations for FreeRTOS

There's currently no well-defined approach to redefining the configs outside of the FreeRTOS source tree. By default, CMake will look for the *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/ and *freertos*/demos/include/ directories. However, you can use a workaround to tell the compiler to search other directories first. For example, you can add another folder for FreeRTOS configurations.

```
- freertos
- freertos-configs
  - aws_clientcredential.h
  - aws_clientcredential_keys.h
  - iot_mqtt_agent_config.h
  - iot_config.h
- components
- src
- CMakeLists.txt
```

The files under `freertos-configs` are copied from the *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/ and *freertos*/demos/include/ directories. Then, in your top level `CMakeLists.txt` file, add this line before `add_subdirectory(freertos)` so that the compiler will search this directory first.

```
include_directories(BEFORE freertos-configs)
```

Providing your own sdkconfig for ESP-IDF

In case you want to provide your own `sdkconfig.default`, you can set the CMake variable `IDF_SDKCONFIG_DEFAULTS`, from the command line:

```
cmake -S . -B build-directory -DIDF_SDKCONFIG_DEFAULTS=path_to_your_sdkconfig_defaults
 -DCMAKE_TOOLCHAIN_FILE=freertos/tools/cmake/toolchains/xtensa-esp32.cmake -GNinja
```

If you don't specify a location for your own `sdkconfig.default` file, FreeRTOS uses the default file located at *`freertos`*`/vendors/espressif/boards/esp32/aws_demos/sdkconfig.defaults`.

For more information, see Project Configuration in the Espressif *API Reference* and, if you encounter issues after you have successfully compiled, see the section on Deprecated options and their replacements on that page.

Summary

If you have a project with a component called `example_component`, and you want to override some configurations, here's a complete example of the top level `CMakeLists.txt` file.

```
cmake_minimum_required(VERSION 3.13)

project(freertos_examples)

set(IDF_PROJECT_EXECUTABLE my_app)
set(IDF_EXECUTABLE_SRCS "src/main.c")

# Tell IDF build to link against this target.
set(IDF_PROJECT_EXECUTABLE my_app)

# Add some extra components. IDF_EXTRA_COMPONENT_DIRS is a variable used by ESP-IDF
# to collect extra components.
get_filename_component(
    EXTRA_COMPONENT_DIRS
    "components/example_component" ABSOLUTE
)
list(APPEND IDF_EXTRA_COMPONENT_DIRS ${EXTRA_COMPONENT_DIRS})

# Override the configurations for FreeRTOS.
include_directories(BEFORE freertos-configs)
```

```
# Add FreeRTOS as a subdirectory. AFR_BOARD tells which board to target.
set(AFR_BOARD espressif.esp32_devkitc CACHE INTERNAL "")
add_subdirectory(freertos)

# Link against the mqtt library so that we can use it. Dependencies are transitively
# linked.
target_link_libraries(my_app PRIVATE AFR::core_mqtt)
```

Troubleshooting

- If you're running macOS and the operating system doesn't recognize your ESP-WROVER-KIT, make sure you don't have the D2XX drivers installed. To uninstall them, follow the instructions in the FTDI Drivers Installation Guide for macOS X.
- The monitor utility provided by ESP-IDF (and invoked using make monitor) helps you decode addresses. For this reason, it can help you get some meaningful backtraces in the event the application stops working. For more information, see Automatic Address Decoding on the Espressif website.
- It's also possible to enable GDBstub for communication with gdb without requiring any special JTAG hardware. For more information, see Launching GDB with GDBStub on the Espressif website.
- For information about setting up an OpenOCD-based environment if JTAG hardware-based debugging is required, see JTAG Debugging on the Espressif website.
- If `pyserial` can't be installed using `pip` on macOS, download it from the pyserial website.
- If the board resets continuously, try erasing the flash by entering the following command on the terminal.

  ```
  make erase_flash
  ```

- If you see errors when you run `idf_monitor.py`, use Python 2.7.
- Required libraries from ESP-IDF are included in FreeRTOS, so there is no need to download them externally. If the `IDF_PATH` environment variable is set, we recommend that you clear it before you build FreeRTOS.
- On Windows, it can take 3-4 minutes for the project to build. To reduce the build time, you can use the `-j4` switch on the make command.

  ```
  make flash monitor -j4
  ```

- If your device has trouble connecting to AWS IoT, open the `aws_clientcredential.h` file, and verify that the configuration variables are properly defined in the file. `clientcredentialMQTT_BROKER_ENDPOINT[]` should look like `1234567890123-ats.iot.us-east-1.amazonaws.com`.
- If you're following the steps in Using FreeRTOS in your own CMake project for ESP32 and you see undefined reference errors from the linker, it's usually because of missing dependent libraries or demos. To add them, update the `CMakeLists.txt` file (under the root directory) using the standard CMake function `target_link_libraries`.
- ESP-IDF v4.2 supports the use of the *xtensa\-esp32\-elf\-gcc 8\.2\.0\.* toolchain. If you're using an earlier version of the Xtensa toolchain, download the required version.
- If you see an error log like the following about python dependencies that are not being met for ESP-IDF v4.2:

```
The following Python requirements are not satisfied:
  click>=5.0
  pyserial>=3.0
  future>=0.15.2
  pyparsing>=2.0.3,<2.4.0
  pyelftools>=0.22
  gdbgui==0.13.2.0
  pygdbmi<=0.9.0.2
  reedsolo>=1.5.3,<=1.5.4
  bitstring>=3.1.6
  ecdsa>=0.16.0
  Please follow the instructions found in the "Set up the tools" section of ESP-IDF
 Getting Started Guide
```

  Install the python dependencies on your platform using the following Python command:

```
root/vendors/espressif/esp-idf/requirements.txt
```

For more troubleshooting information, see Troubleshooting getting started.

Debugging

Debugging code on Espressif ESP32-DevKitC and ESP-WROVER-KIT (ESP-IDF v4.2)

This section shows you how to debug Espressif hardware using ESP-IDF v4.2. You need a JTAG to USB cable. We use a USB to MPSSE cable (for example, the FTDI C232HM-DDHSL-0).

ESP-DevKitC JTAG setup

For the FTDI C232HM-DDHSL-0 cable, these are the connections to the ESP32 DevkitC.

C232HM-DDHSL-0 Wire Color	ESP32 GPIO Pin	JTAG Signal Name
Brown (pin 5)	IO14	TMS
Yellow (pin 3)	IO12	TDI
Black (pin 10)	GND	GND
Orange (pin 2)	IO13	TCK
Green (pin 4)	IO15	TDO

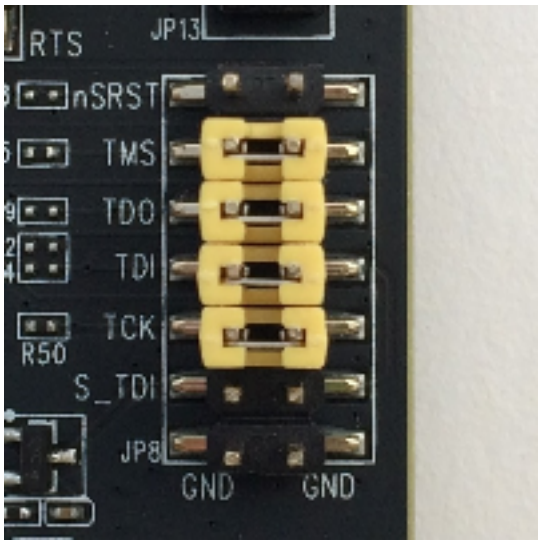
ESP-WROVER-KIT JTAG setup

For the FTDI C232HM-DDHSL-0 cable, these are the connections to the ESP32-WROVER-KIT.

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

These tables were developed from the FTDI C232HM-DDHSL-0 datasheet. For more information, see the section "C232HM MPSSE Cable Connection and Mechanical Details in the data sheet.

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



Debugging on Windows (ESP-IDF v4.2)

To set up for debugging on Windows

1. Connect the USB side of the FTDI C232HM-DDHSL-0 to your computer and the other side as described in Debugging code on Espressif ESP32-DevKitC and ESP-WROVER-KIT (ESP-IDF v4.2). The FTDI C232HM-DDHSL-0 device should appear in **Device Manager** under **Universal Serial Bus Controllers**.
2. Under the list of universal serial bus devices, right-click the **C232HM-DDHSL-0** device, and then choose **Properties**.

   > **Note**
   >
   > The device might be listed as **USB Serial Port**.

   To see the properties of the device, in the properties window, choose the **Details** tab. If the device isn't listed, install the Windows driver for FTDI C232HM-DDHSL-0.
3. On the **Details** tab, choose **Property**, and then choose **Hardware IDs**. You should see something like this in the **Value** field.

   ```
   FTDIBUS\COMPORT&VID_0403&PID_6014
   ```

   In this example, the vendor ID is 0403 and the product ID is 6014.

   Verify these IDs match the IDs in `projects/espressif/esp32/make/aws_demos/esp32_devkitj_v1.cfg`. The IDs are specified in a line that begins with `ftdi_vid_pid` followed by a vendor ID and a product ID.

   ```
   ftdi_vid_pid 0x0403 0x6014
   ```
4. Download OpenOCD for Windows.
5. Unzip the file to `C:\` and add `C:\openocd-esp32\bin` to your system path.
6. OpenOCD requires libusb, which is not installed by default on Windows. To install libusb:

   a. Download zadig.exe.

   b. Run `zadig.exe`. From the **Options** menu, choose **List All Devices**.

c. From the dropdown menu, choose **C232HM-DDHSL-0**.

d. In the target driver field, to the right of the green arrow, choose **WinUSB**.

e. For the list under the target driver field, choose the arrow, and then choose **Install Driver**. Choose **Replace Driver**.

7. Open a command prompt, navigate to the root of your FreeRTOS download directory, and run the following command.

```
idf.py openocd
```

Leave this command prompt open.

8. Open a new command prompt, navigate to the root of your FreeRTOS download directory, and run

```
idf.py flash monitor
```

9. Open another command prompt, navigate to the root of your FreeRTOS download directory, and wait until the demo starts running on your board. When it does, run

```
idf.py gdb
```

The program should stop in the `main` function.

> **Note**
>
> The ESP32 supports a maximum of two break points.

Debugging on macOS (ESP-IDF v4.2)

1. Download the FTDI driver for macOS.
2. Download OpenOCD.
3. Extract the downloaded .tar file and set the path in `.bash_profile` to `OCD_INSTALL_DIR/openocd-esp32/bin`.
4. Use the following command to install `libusb` on macOS.

```
brew install libusb
```

5. Use the following command to unload the serial port driver.

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

6. Use the following command to unload the serial port driver.

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

7. If you're running a macOS version later than 10.9, use the following command to unload the Apple FTDI driver.

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

8. Use the following command to get the product ID and vendor ID of the FTDI cable. It lists the attached USB devices.

```
system_profiler SPUSBDataType
```

   The output from `system_profiler` should look like the following.

```
   DEVICE:

   Product ID: product-ID
   Vendor ID: vendor-ID (Future Technology Devices International Limited)
```

9. Open the `projects/espressif/esp32/make/aws_demos/esp32_devkitj_v1.cfg` file. The vendor ID and product ID for your device are specified in a line that begins with `ftdi_vid_pid`. Change the IDs to match the IDs from the `system_profiler` output in the previous step.

10. Open a terminal window, navigate to the root of your FreeRTOS download directory, and use the following command to run OpenOCD.

```
idf.py openocd
```

    Leave this terminal window open.

11. Open a new terminal, and use the following command to load the FTDI serial port driver.

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

12. Navigate to the root of your FreeRTOS download directory, and run

```
idf.py flash monitor
```

13. Open another new terminal, navigate to the root of your FreeRTOS download directory, and run

```
idf.py gdb
```

The program should stop at `main`.

Debugging on Linux (ESP-IDF v4.2)

1. Download OpenOCD. Extract the tarball and follow the installation instructions in the readme file.
2. Use the following command to install libusb on Linux.

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

3. Open a terminal and enter **ls -l /dev/ttyUSB*** to list all USB devices connected to your computer. This helps you check if the board's USB ports are recognized by the operating system. You should see output like the following.

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

4. Sign off and then sign in and cycle the power to the board to make the changes take effect. In a terminal prompt, list the USB devices. Make sure the group owner has changed from `dialout` to `plugdev`.

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

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

5. In a terminal window, navigate to the root of your FreeRTOS download directory, and use the following command to run OpenOCD.

```
idf.py openocd
```

6. Open another terminal, navigate to the root of your FreeRTOS download directory, and run the following command.

```
idf.py flash monitor
```

7. Open another terminal, navigate the root of your FreeRTOS download directory, and run the following command:

```
idf.py gdb
```

The program should stop in `main()`.

Getting started with the Espressif ESP32-WROOM-32SE

Important

This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Note

- To explore how to integrate FreeRTOS modular libraries and demos within your own Espressif IDF project, see our featured reference integration for ESP32-C3 platform.
- Currently, the FreeRTOS port for ESP32-WROOM-32SE doesn't support the symmetric multiprocessing (SMP) feature.

This tutorial shows you how to get started with the Espressif ESP32-WROOM-32SE. To purchase one from our partner on the AWS Partner Device catalog, see ESP32-WROOM-32SE.

Overview

This tutorial guides you through the following steps:

1. Connect your board to a host machine.
2. Install software on your host machine to develop and debug embedded applications for your microcontroller board.
3. Cross compile a FreeRTOS demo application to a binary image.
4. Load the application binary image to your board, and then run the application.
5. Monitor and debug the running application by using a serial connection.

Prerequisites

Before you get started with FreeRTOS on your Espressif board, you must set up your AWS account and permissions.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Get started

> **Note**
>
> The Linux commands in this tutorial require that you use the Bash shell.

1. **Set up the Espressif hardware.**

   For information about setting up the ESP32-WROOM-32SE development board hardware, see the ESP32-DevKitC V4 Getting Started Guide.

> **Important**
>
> When you reach the **Installation Step by Step** section of the guide, follow till you complete Step 4 (Set up the environment variables). Stop after you complete Step 4 and follow the remaining steps here.

2. Download Amazon FreeRTOS from GitHub. (For instructions, see the README.md file.)
3. **Set up your development environment**.

   To communicate with your board, you must install a toolchain. Espressif provides the ESP-IDF to develop software for their boards. Since the ESP-IDF has its own version of the FreeRTOS Kernel integrated as a component, Amazon FreeRTOS includes a custom version of the ESP-IDF v4.2 that has the FreeRTOS Kernel removed. This fixes problems with duplicate files when you compile. To use the custom version of the ESP-IDF v4.2 included with Amazon FreeRTOS, follow the instructions below for your host machine's operating system.

   Windows

   1. Download ESP-IDF's Universal Online Installer for Windows.
   2. Run the **Universal Online Installer**.
   3. When you get to the step **Download or use ESP-IDF**, select **Use an existing ESP-IDF directory** and set **Choose existing ESP-IDF directory** to *`freertos`*`/vendors/espressif/esp-idf`.
   4. Complete the installation.

   macOS

   1. Follow the instructions in the Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for macOS.

      > **Important**
      >
      > When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop, and then return to the instructions on this page.

   2. Open a command line window.

3. Navigate to the FreeRTOS download directory, and then run the following script to download and install the espressif toolchain for your platform.

```
vendors/espressif/esp-idf/install.sh
```

4. Add the ESP-IDF toolchain tools to your terminal's path with the following command.

```
source vendors/espressif/esp-idf/export.sh
```

Linux

1. Follow the instructions in the Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for Linux.

> **Important**
>
> When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop, and then return to the instructions on this page.

2. Open a command line window.
3. Navigate to the FreeRTOS download directory, and then run the following script to download and install the Espressif toolchain for your platform.

```
vendors/espressif/esp-idf/install.sh
```

4. Add the ESP-IDF toolchain tools to your terminal's path with the following command.

```
source vendors/espressif/esp-idf/export.sh
```

4. **Establish a serial connection.**

a. To establish a serial connection between your host machine and the ESP32-WROOM-32SE, install the CP210x USB to UART Bridge VCP drivers. You can download these drivers from Silicon Labs.

b. Follow the steps to Establish a Serial Connection with ESP32.

c. After you establish a serial connection, make a note of the serial port for your board's connection. You need it to flash the demo.

Configure the FreeRTOS demo applications

For this tutorial, the FreeRTOS configuration file is located at *freertos*/vendors/espressif/boards/*board-name*/aws_demos/config_files/FreeRTOSConfig.h. (For example, if AFR_BOARD espressif.esp32_devkitc is chosen, the configuration file is located at *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/FreeRTOSConfig.h.)

> **Important**
>
> The ATECC608A device has a one-time initialization that is locked onto the device the first time a project is run (during the call to C_InitToken). However, the FreeRTOS demo project and test project have different configurations. If the device is locked during the demo project configurations, not all tests in the test project will succeed.

1. Configure the FreeRTOS Demo Project by following the steps in Configuring the FreeRTOS demos. When you get to the last step **To format your AWS IoT credentials**, stop, and perform the following steps.
2. Microchip has provided several scripting tools to help with the setup of the ATECC608A parts. Navigate to the *freertos*/vendors/microchip/example_trust_chain_tool directory, and open the README.md file.
3. To provision your device, follow the instructions in the README.md file. The steps include the following:
    1. Create and register a certificate authority with AWS.
    2. Generate your keys on the ATECC608A and export the public key and device serial number.
    3. Generate a certificate for the device and register that certificate with AWS.
4. Load the CA certificate and device certificate onto the device by following the instructions for Developer-mode key provisioning.

Monitoring MQTT messages on the AWS Cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT Test Client**.
3. In **Subscription topic**, enter *your-thing-name*`/example/topic` and then choose **Subscribe to topic**.

Build, flash, and run the FreeRTOS demo project using the idf.py script

You can use Espressif's IDF utility (`idf.py`) to generate the build files, build the application binary, and flash the binaries onto your device.

> **Note**
>
> Some setups might require that you use the port option "`-p port-name`" with `idf.py` to specify the correct port, as in the following example.
>
> ```
> idf.py -p /dev/cu.usbserial-00101301B flash
> ```

Build and flash FreeRTOS on Windows, Linux, and macOS (ESP-IDF v4.2)

1. Navigate to the root of your FreeRTOS download directory.
2. In a command line window, enter the following command to add the ESP-IDF tools to your terminal's PATH:

   Windows ("Command" app)

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Windows ("ESP-IDF 4.x CMD" app)

   (This has already been done when you opened the app.)

   Linux / macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

3. Configure cmake in the `build` directory and build the firmware image with the following command.

```
idf.py -DVENDOR=espressif -DBOARD=esp32_ecc608a_devkitc -DCOMPILER=xtensa-esp32
 build
```

You should see output like this following example.

```
   Running cmake in directory /path/to/hello_world/build
   Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
   Warn about uninitialized values.
   -- Found Git: /usr/bin/git (found version "2.17.0")
   -- Building empty aws_iot component due to configuration
   -- Component names: ...
   -- Component paths: ...

   ... (more lines of build system output)

   [527/527] Generating hello-world.bin
   esptool.py v2.3.1

   Project build complete. To flash, run this command:
   ../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600
 write_flash --flash_mode dio --flash_size detect --flash_freq 40m 0x10000 build/
hello-world.bin  build 0x1000 build/bootloader/bootloader.bin 0x8000 build/
partition_table/partition-table.bin
   or run 'idf.py -p PORT flash'
```

If there are no errors, the build will generate the firmware binary .bin files.

4. Erase your development board's flash memory with the following command.

```
idf.py erase_flash
```

5. Use the `idf.py` script to flash the application binary to your board.

```
idf.py flash
```

6. Monitor the output from your board's serial port with the following command.

```
idf.py monitor
```

Note

- You can combine these commands as in the following example.

```
idf.py erase_flash flash monitor
```

- For certain host machine setups, you must specify the port when you flash the board as in the following example.

```
idf.py erase_flash flash monitor -p /dev/ttyUSB1
```

Build and Flash FreeRTOS with CMake

Besides using the `idf.py` script provided by the IDF SDK to build and run your code, you can also build the project with CMake. Currently it supports Unix Makefile and the Ninja build system.

To build and flash the project

1. In a command line window, navigate to the root of your FreeRTOS download directory.
2. Run the following script to add the ESP-IDF tools to your shell's PATH.

   Windows

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Linux / macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

3. Enter the following command to generate the build files.

   With Unix Makefiles

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32_plus_ecc608a_devkitc -DCOMPILER=xtensa-
   esp32 -S . -B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -
   DAFR_ENABLE_TESTS=0
   ```

With Ninja

```
cmake -DVENDOR=espressif -DBOARD=esp32_plus_ecc608a_devkitc -DCOMPILER=xtensa-
esp32 -S . -B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -
DAFR_ENABLE_TESTS=0 -GNinja
```

4. Erase the flash and then flash the board.

With Unix Makefiles

```
make -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
make -C ./YOUR_BUILD_DIRECTORY flash
```

With Ninja

```
ninja -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
ninja -C ./YOUR_BUILD_DIRECTORY flash
```

Additional information

For more information about using and troubleshooting Espressif ESP32 boards, see the following topics:

- Using FreeRTOS in your own CMake project for ESP32
- Troubleshooting
- Debugging

Getting started with the Espressif ESP32-S2

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you

already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Note

To explore how to integrate FreeRTOS modular libraries and demos within your own Espressif IDF project, see our featured reference integration for ESP32-C3 platform.

This tutorial shows you how to get started with the Espressif ESP32-S2 SoC and ESP32-S2-Saola-1 development boards.

Overview

This tutorial guides you through the following steps:

1. Connect your board to a host machine.
2. Install software on your host machine to develop and debug embedded applications for your microcontroller board.
3. Cross-compile a FreeRTOS demo application to a binary image.
4. Load the application binary image to your board, and then run the application.
5. Monitor and debug the running application using a serial connection.

Prerequisites

Before you get started with FreeRTOS on your Espressif board, you must set up your AWS account and permissions.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.

- (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Get started

> **Note**
>
> The Linux commands in this tutorial require that you use the Bash shell.

1. **Set up the Espressif hardware.**

   For information about setting up the ESP32-S2 development board hardware, see the ESP32-S2-Saola-1 Getting Started Guide.

   > **Important**
   >
   > When you reach the **Get Started** section of the Espressif guides, stop, and then return to the instructions on this page.

2. Download Amazon FreeRTOS from GitHub. (For instructions, see the README.md file.)
3. **Set up your development environment**.

   To communicate with your board, you must install a toolchain. Espressif provides the ESP-IDF to develop software for their boards. Since the ESP-IDF has its own version of the FreeRTOS Kernel integrated as a component, Amazon FreeRTOS includes a custom version of the ESP-IDF v4.2 that has the FreeRTOS Kernel removed. This fixes problems with duplicate files when you compile. To use the custom version of the ESP-IDF v4.2 included with Amazon FreeRTOS, follow the instructions below for your host machine's operating system.

   Windows

   1. Download ESP-IDF's Universal Online Installer for Windows.
   2. Run the **Universal Online Installer**.
   3. When you get to the step **Download or use ESP-IDF**, select **Use an existing ESP-IDF directory** and set **Choose existing ESP-IDF directory** to *`freertos`*`/vendors/espressif/esp-idf`.
   4. Complete the installation.

macOS

1. Follow the instructions in the Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for macOS.

   > **Important**
   >
   > When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop, and then return to the instructions on this page.

2. Open a command line window.
3. Navigate to the FreeRTOS download directory, and then run the following script to download and install the espressif toolchain for your platform.

   ```
   vendors/espressif/esp-idf/install.sh
   ```

4. Add the ESP-IDF toolchain tools to your terminal's path with the following command.

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

Linux

1. Follow the instructions in the Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for Linux.

   > **Important**
   >
   > When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop, and then return to the instructions on this page.

2. Open a command line window.
3. Navigate to the FreeRTOS download directory, and then run the following script to download and install the Espressif toolchain for your platform.

   ```
   vendors/espressif/esp-idf/install.sh
   ```

4. Add the ESP-IDF toolchain tools to your terminal's path with the following command.

```
source vendors/espressif/esp-idf/export.sh
```

4. **Establish a serial connection.**

   a. To establish a serial connection between your host machine and the ESP32-DevKitC, install the CP210x USB to UART Bridge VCP drivers. You can download these drivers from Silicon Labs.

   b. Follow the steps to Establish a Serial Connection with ESP32.

   c. After you establish a serial connection, make a note of the serial port for your board's connection. You need it to flash the demo.

Configure the FreeRTOS demo applications

For this tutorial, the FreeRTOS configuration file is located at *freertos*`/vendors/espressif/boards/`*board-name*`/aws_demos/config_files/FreeRTOSConfig.h`. (For example, if `AFR_BOARD espressif.esp32_devkitc` is chosen, the configuration file is located at *freertos*`/vendors/espressif/boards/esp32/aws_demos/config_files/FreeRTOSConfig.h`.)

1. If you're running macOS or Linux, open a terminal prompt. If you're running Windows, open the "ESP-IDF 4.x CMD" app (if you included this option when you installed the ESP-IDF toolchain), or the "Command Prompt" app otherwise.
2. To verify that you have Python3 installed, run the following:

   ```
   python --version
   ```

   The version installed is displayed. If you don't have Python 3.0.1 or later installed, you can install it from the Python website.
3. You need the AWS Command Line Interface (CLI) to run AWS IoT commands. If you're running Windows, use the `easy_install awscli` command to install the AWS CLI in the "Command" or "ESP-IDF 4.x CMD" app.

   If you're running macOS or Linux, see Installing the AWS CLI.
4. Run

   ```
   aws configure
   ```

and configure the AWS CLI with your AWS access key ID, secret access key, and default AWS Region. For more information, see Configuring the AWS CLI.

5. Use the following command to install the AWS SDK for Python (boto3):

   - On Windows, in the "Command" or "ESP-IDF 4.x CMD" app, run

     ```
     easy_install boto3
     ```

   - On macOS or Linux, run

     ```
     pip install tornado nose --user
     ```

     and then run

     ```
     pip install boto3 --user
     ```

FreeRTOS includes the `SetupAWS.py` script to make it easier to set up your Espressif board to connect to AWS IoT.

To run the configuration script

1. To configure the script, open *`freertos`*`/tools/aws_config_quick_start/configure.json` and set the following attributes:

   `afr_source_dir`

   The complete path to the *`freertos`* directory on your computer. Make sure that you use forward slashes to specify this path.

   `thing_name`

   The name that you want to assign to the AWS IoT thing that represents your board.

   `wifi_ssid`

   The SSID of your Wi-Fi network.

   `wifi_password`

   The password for your Wi-Fi network.

`wifi_security`

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

- `eWiFiSecurity0pen` (Open, no security)
- `eWiFiSecurityWEP` (WEP security)
- `eWiFiSecurityWPA` (WPA security)
- `eWiFiSecurityWPA2` (WPA2 security)

2. If you're running macOS or Linux, open a terminal prompt. If you're running Windows, open the "ESP-IDF 4.x CMD" or "Command" app.
3. Navigate to the *`freertos`*`/tools/aws_config_quick_start` directory and run

```
python SetupAWS.py setup
```

The script does the following:

- Creates an AWS IoT thing, certificate, and policy.
- Attaches the AWS IoT policy to the certificate and the certificate to the AWS IoT thing.
- Populates the `aws_clientcredential.h` file with your AWS IoT endpoint, Wi-Fi SSID, and credentials.
- Formats your certificate and private key and writes them to the `aws_clientcredential_keys.h` header file.

> **Note**
>
> The certificate is hardcoded for demonstration purposes only. Production-level applications should store these files in a secure location.

For more information about `SetupAWS.py`, see the `README.md` in the *`freertos`*`/tools/aws_config_quick_start` directory.

Monitoring MQTT messages on the AWS Cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT Test Client**.
3. In **Subscription topic**, enter *your-thing-name*`/example/topic`, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build, flash, and run the FreeRTOS demo project using the idf.py script

You can use Espressif's IDF utility to generate the build files, build the application binary, and flash your board.

Build and flash FreeRTOS on Windows, Linux, and macOS (ESP-IDF v4.2)

Use the `idf.py` script to build the project and flash the binaries onto your device.

> **Note**
>
> Some setups might require that you use the port option `-p port-name` with `idf.py` to specify the correct port, as in the following example.
>
> ```
> idf.py -p /dev/cu.usbserial-00101301B flash
> ```

To build and flash the project

1. Navigate to the root of your FreeRTOS download directory.
2. In a command line window, enter the following command to add the ESP-IDF tools to your terminal's PATH:

   Windows ("Command" app)

   ```
   vendors\espressif\esp-idf\export.bat
   ```

Windows ("ESP-IDF 4.x CMD" app)

(This has already been done when you opened the app.)

Linux / macOS

```
source vendors/espressif/esp-idf/export.sh
```

3. Configure cmake in the `build` directory and build the firmware image with the following command.

```
idf.py -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 build
```

You should see output like this following example.

```
Executing action: all (aliases: build)
   Running cmake in directory /path/to/hello_world/build
   Executing "cmake -G Ninja -DPYTHON_DEPS_CHECKED=1 -DESP_PLATFORM=1
 -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 -
DCCACHE_ENABLE=0 /path/to/hello_world"...
   -- The C compiler identification is GNU 8.4.0
   -- The CXX compiler identification is GNU 8.4.0
   -- The ASM compiler identification is GNU

   ... (more lines of build system output)

   [1628/1628] Generating binary image from built executable
   esptool.py v3.0
   Generated /path/to/hello_world/build/aws_demos.bin

   Project build complete. To flash, run this command:
   esptool.py -p (PORT) -b 460800 --before default_reset --after hard_reset --chip
 esp32s2  write_flash --flash_mode dio --flash_size detect --flash_freq 80m 0x1000
 build/bootloader/bootloader.bin 0x8000 build/partition_table/partition-table.bin
 0x16000 build/ota_data_initial.bin 0x20000 build/aws_demos.bin
   or run 'idf.py -p (PORT) flash'
```

If there are no errors, the build generates the firmware binary .bin files.

4. Erase your development board's flash memory with the following command.

```
idf.py erase_flash
```

5. Use the `idf.py` script to flash the application binary to your board.

```
idf.py flash
```

6. Monitor the output from your board's serial port with the following command.

```
idf.py monitor
```

Note

- You can combine these commands as in the following example.

```
idf.py erase_flash flash monitor
```

- For certain host machine setups, you must specify the port when you flash the board as in the following example.

```
idf.py erase_flash flash monitor -p /dev/ttyUSB1
```

Build and Flash FreeRTOS with CMake

Besides using the `idf.py` script provided by the IDF SDK to build and run your code, you can also build the project with CMake. Currently it supports Unix Makefile and the Ninja build system.

To build and flash the project

1. In a command line window, navigate to the root of your FreeRTOS download directory.
2. Run the following script to add the ESP-IDF tools to your shell's PATH.

- Windows

```
vendors\espressif\esp-idf\export.bat
```

- Linux / macOS

```
source vendors/espressif/esp-idf/export.sh
```

3. Enter the following command to generate the build files.

   - With Unix Makefiles

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 -S . -
   B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0
   ```

   - With Ninja

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 -S . -
   B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0 -GNinja
   ```

4. Build the project.

   - With Unix Makefiles

   ```
   make -C ./YOUR_BUILD_DIRECTORY -j8
   ```

   - With Ninja

   ```
   ninja -C ./YOUR_BUILD_DIRECTORY -j8
   ```

5. Erase the flash and then flash the board.

   - With Unix Makefiles

   ```
   make -C ./YOUR_BUILD_DIRECTORY erase_flash
   ```

   ```
   make -C ./YOUR_BUILD_DIRECTORY flash
   ```

   - With Ninja

   ```
   ninja -C ./YOUR_BUILD_DIRECTORY erase_flash
   ```

   ```
   ninja -C ./YOUR_BUILD_DIRECTORY flash
   ```

Additional information

For more information about using and troubleshooting Espressif ESP32 boards, see the following topics:

- Using FreeRTOS in your own CMake project for ESP32
- Troubleshooting
- Debugging

Getting started with the Infineon XMC4800 IoT Connectivity Kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Infineon XMC4800 IoT Connectivity Kit. If you do not have the Infineon XMC4800 IoT Connectivity Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

If you want to open a serial connection with the board to view logging and debugging information, you need a 3.3V USB/Serial converter, in addition to the XMC4800 IoT Connectivity Kit. The CP2104 is a common USB/Serial converter that is widely available in boards such as Adafruit's CP2104 Friend.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains instructions for the following getting started steps:

1. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.

2. Cross compiling a FreeRTOS demo application to a binary image.
3. Loading the application binary image to your board, and then running the application.
4. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Set up your development environment

FreeRTOS uses Infineon's DAVE development environment to program the XMC4800. Before you begin, you need to download and install DAVE and some J-Link drivers to communicate with the on-board debugger.

Install DAVE

1. Go to Infineon's DAVE software download page.
2. Choose the DAVE package for your operating system and submit your registration information. After registering with Infineon, you should receive a confirmation email with a link to download a .zip file.
3. Download the DAVE package .zip file (`DAVE_version_os_date.zip`), and unzip it to the location where you want to install DAVE (for example, `C:\DAVE4`).

   > **Note**
   >
   > Some Windows users have reported problems using Windows Explorer to unzip the file. We recommend that you use a third-party program such as 7-Zip.

4. To launch DAVE, run the executable file found in the unzipped `DAVE_version_os_date.zip` folder.

For more information, see the DAVE Quick Start Guide.

Install Segger J-Link drivers

To communicate with the XMC4800 Relax EtherCAT board's on-board debugging probe, you need the drivers included in the J-Link Software and Documentation pack. You can download the J-Link Software and Documentation pack from Segger's J-Link software download page.

Establish a serial connection

Setting up a serial connection is optional, but recommended. A serial connection allows your board to send logging and debugging information in a form that you can view on your development machine.

The XMC4800 demo application uses a UART serial connection on pins P0.0 and P0.1, which are labeled on the XMC4800 Relax EtherCAT board's silkscreen. To set up a serial connection:

1. Connect the pin labeled “RX<P0.0” to your USB/Serial converter's “TX” pin.
2. Connect the pin labeled “TX>P0.1” to your USB/Serial converter's “RX” pin.
3. Connect your serial converter's Ground pin to one of the pins labeled “GND” on your board. The devices must share a common ground.

Power is supplied from the USB debugging port, so do not connect your serial adapter's positive voltage pin to the board.

> **Note**
>
> Some serial cables use a 5V signaling level. The XMC4800 board and the Wi-Fi Click module require a 3.3V. Do not use the board's IOREF jumper to change the board's signals to 5V.

With the cable connected, you can open a serial connection on a terminal emulator such as GNU Screen. The baud rate is set to 115200 by default with 8 data bits, no parity, and 1 stop bit.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run the FreeRTOS demo project

Import the FreeRTOS demo into DAVE

1. Start DAVE.
2. In DAVE, choose **File**, **Import**. In the **Import** window, expand the **Infineon** folder, choose **DAVE Project**, and then choose **Next**.
3. In the **Import DAVE Projects** window, choose **Select Root Directory**, choose **Browse**, and then choose the XMC4800 demo project.

   In the directory where you unzipped your FreeRTOS download, the demo project is located in `projects/infineon/xmc4800_iotkit/dave4/aws_demos`.

   Make sure that **Copy Projects Into Workspace** is unchecked.
4. Choose **Finish**.

   The `aws_demos` project should be imported into your workspace and activated.
5. From the **Project** menu, choose **Build Active Project**.

   Make sure that the project builds without errors.

Run the FreeRTOS demo project

1. Use a USB cable to connect your XMC4800 IoT Connectivity Kit to your computer. The board has two microUSB connectors. Use the one labeled “X101”, where Debug appears next to it on the board's silkscreen.
2. From the **Project** menu, choose **Rebuild Active Project** to rebuild `aws_demos` and ensure that your configuration changes are picked up.
3. From **Project Explorer**, right-click `aws_demos`, choose **Debug As**, and then choose **DAVE C/C++ Application**.
4. Double-click **GDB SEGGER J-Link Debugging** to create a debug confirmation. Choose **Debug**.
5. When the debugger stops at the breakpoint in `main()`, from the **Run** menu, choose **Resume**.

In the AWS IoT console, the MQTT client from steps 4-5 should display the MQTT messages sent by your device. If you use the serial connection, you see something like this on the UART output:

```
0 0 [Tmr Svc] Starting key provisioning...
1 1 [Tmr Svc] Write root certificate...
2 4 [Tmr Svc] Write device private key...
3 82 [Tmr Svc] Write device certificate...
4 86 [Tmr Svc] Key provisioning done...
5 291 [Tmr Svc] Wi-Fi module initialized. Connecting to AP...
.6 8046 [Tmr Svc] Wi-Fi Connected to AP. Creating tasks which use network...
7 8058 [Tmr Svc] IP Address acquired [IP Address]
8 8058 [Tmr Svc] Creating MQTT Echo Task...
9 8059 [MQTTEcho] MQTT echo attempting to connect to [MQTT Broker].
...10 23010 [MQTTEcho] MQTT echo connected.
11 23010 [MQTTEcho] MQTT echo test echoing task created.
.12 26011 [MQTTEcho] MQTT Echo demo subscribed to iotdemo/#
13 29012 [MQTTEcho] Echo successfully published 'Hello World 0'
.14 32096 [Echoing] Message returned with ACK: 'Hello World 0 ACK'
.15 37013 [MQTTEcho] Echo successfully published 'Hello World 1'
16 40080 [Echoing] Message returned with ACK: 'Hello World 1 ACK'
.17 45014 [MQTTEcho] Echo successfully published 'Hello World 2'
.18 48091 [Echoing] Message returned with ACK: 'Hello World 2 ACK'
.19 53015 [MQTTEcho] Echo successfully published 'Hello World 3'
.20 56087 [Echoing] Message returned with ACK: 'Hello World 3 ACK'
.21 61016 [MQTTEcho] Echo successfully published 'Hello World 4'
22 64083 [Echoing] Message returned with ACK: 'Hello World 4 ACK'
.23 69017 [MQTTEcho] Echo successfully published 'Hello World 5'
.24 72091 [Echoing] Message returned with ACK: 'Hello World 5 ACK'
.25 77018 [MQTTEcho] Echo successfully published 'Hello World 6'
26 80085 [Echoing] Message returned with ACK: 'Hello World 6 ACK'
.27 85019 [MQTTEcho] Echo successfully published 'Hello World 7'
.28 88086 [Echoing] Message returned with ACK: 'Hello World 7 ACK'
.29 93020 [MQTTEcho] Echo successfully published 'Hello World 8'
.30 96088 [Echoing] Message returned with ACK: 'Hello World 8 ACK'
.31 101021 [MQTTEcho] Echo successfully published 'Hello World 9'
32 104102 [Echoing] Message returned with ACK: 'Hello World 9 ACK'
.33 109022 [MQTTEcho] Echo successfully published 'Hello World 10'
.34 112047 [Echoing] Message returned with ACK: 'Hello World 10 ACK'
.35 117023 [MQTTEcho] Echo successfully published 'Hello World 11'
36 120089 [Echoing] Message returned with ACK: 'Hello World 11 ACK'
.37 122068 [MQTTEcho] MQTT echo demo finished.
38 122068 [MQTTEcho] ----Demo finished----
```

Build the FreeRTOS demo with CMake

If you prefer not to use an IDE for FreeRTOS development, you can alternatively use CMake to build and run the demo applications or applications that you have developed using third-party code editors and debugging tools.

> **Note**
>
> This section covers using CMake on Windows with MingW as the native build system. For more information about using CMake with other operating systems and options, see Using CMake with FreeRTOS. (MinGW is a minimalist development environment for native Microsoft Windows applications.)

To build the FreeRTOS demo with CMake

1. Set up the GNU Arm Embedded Toolchain.

   a. Download a Windows version of the toolchain from the Arm Embedded Toolchain download page.

   > **Note**
   >
   > We recommend that you download a version other than "8-2018-q4-major", due to a bug reported with the “objcopy” utility in that version.

   b. Open the downloaded toolchain installer, and follow the installation wizard's instructions to install the toolchain.

   > **Important**
   >
   > On the final page of the installation wizard, select **Add path to environment variable** to add the toolchain path to the system path environment variable.

2. Install CMake and MingW.

   For instructions, see CMake Prerequisites.

3. Create a folder to contain the generated build files (*`build-folder`*).

4. Change directories to your FreeRTOS download directory (*freertos*), and use the following command to generate the build files:

```
cmake -DVENDOR=infineon -DBOARD=xmc4800_iotkit -DCOMPILER=arm-gcc -S . -B build-
folder -G "MinGW Makefiles" -DAFR_ENABLE_TESTS=0
```

5. Change directories to the build directory (*build-folder*), and use the following command to build the binary:

```
cmake --build . --parallel 8
```

   This command builds the output binary `aws_demos.hex` to the build directory.

6. Flash and run the image with JLINK.

   a. From the build directory (*build-folder*), use the following commands to create a flash script:

```
echo loadfile aws_demos.hex > flash.jlink
```

```
echo r >> flash.jlink
```

```
echo g >> flash.jlink
```

```
echo q >> flash.jlink
```

   b. Flash the image using the JLNIK executable.

```
JLINK_PATH\JLink.exe  -device XMC4800-2048 -if SWD -speed auto -CommanderScript
 flash.jlink
```

   The application logs should be visible through the serial connection that you established with the board.

Troubleshooting

If you haven't already, be sure to configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Infineon OPTIGA Trust X and XMC4800 IoT Connectivity Kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Infineon OPTIGA Trust X Secure Element and XMC4800 IoT Connectivity Kit. In comparison to the Getting started with the Infineon XMC4800 IoT Connectivity Kit tutorial, this guide shows you how to provide secure credentials using an Infineon OPTIGA Trust X Secure Element.

You need the following hardware:

1. Host MCU - Infineon XMC4800 IoT Connectivity Kit, visit the AWS Partner Device Catalog to purchase one from our partner.
2. Security Extension Pack:
   - Secure Element - Infineon OPTIGA Trust X.

     Visit the AWS Partner Device Catalog to purchase them from our partner.
   - Personalization Board - Infineon OPTIGA Personalisation Board.
   - Adapter Board - Infineon MyIoT Adapter.

To follow the steps here, you must open a serial connection with the board to view logging and debugging information. (One of the steps requires you to copy a public key from the serial debugging output from the board and paste it to a file.) To do this, you need a 3.3V USB/Serial converter in addition to the XMC4800 IoT Connectivity Kit. The JBtek EL-PN-47310126 USB/Serial converter is known to work for this demo. You also need three male-to-male jumper wires (for receive (RX), transmit (TX), and ground (GND)) to connect the serial cable to the Infineon MyIoT Adapter board.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. For instructions, see Option #2: onboard private key generation. In this tutorial, the path to the FreeRTOS download directory is referred to as *freertos*.

Overview

This tutorial contains the following steps:

1. Install software on the host machine to develop and debug embedded applications for your microcontroller board.
2. Cross-compile a FreeRTOS demo application to a binary image.
3. Load the application binary image to your board, and then run the application.
4. For monitoring and debugging purposes, interact with the application running on your board across a serial connection.

Set up your development environment

FreeRTOS uses Infineon's DAVE development environment to program the XMC4800. Before you begin, download and install DAVE and some J-Link drivers to communicate with the on-board debugger.

Install DAVE

1. Go to Infineon's DAVE software download page.
2. Choose the DAVE package for your operating system and submit your registration information. After you register, you should receive a confirmation email with a link to download a .zip file.
3. Download the DAVE package .zip file (`DAVE_`*`version_os_date`*`.zip`), and unzip it to the location where you want to install DAVE (for example, `C:\DAVE4`).

   > **Note**
   >
   > Some Windows users have reported problems using Windows Explorer to unzip the file. We recommend that you use a third-party program such as 7-Zip.

4. To launch DAVE, run the executable file found in the unzipped `DAVE_`*`version_os_date`*`.zip` folder.

For more information, see the DAVE Quick Start Guide.

Install Segger J-Link drivers

To communicate with the XMC4800 IoT Connectivity kit's on-board debugging probe, you need the drivers included in the J-Link Software and Documentation pack. You can download the J-Link Software and Documentation pack from Segger's J-Link software download page.

Establish a serial connection

Connect the USB/Serial converter cable to the Infineon Shield2Go Adapter. This allows your board to send logging and debugging information in a form that you can view on your development machine. To set up a serial connection:

1. Connect the RX pin to your USB/Serial converter's TX pin.
2. Connect the TX pin to your USB/Serial converter's RX pin.
3. Connect your serial converter's ground pin to one of the GND pins on your board. The devices must share a common ground.

Power is supplied from the USB debugging port, so do not connect your serial adapter's positive voltage pin to the board.

> **Note**
>
> Some serial cables use a 5V signaling level. The XMC4800 board and the Wi-Fi Click module require a 3.3V. Do not use the board's IOREF jumper to change the board's signals to 5V.

With the cable connected, you can open a serial connection on a terminal emulator such as GNU Screen. The baud rate is set to 115200 by default with 8 data bits, no parity, and 1 stop bit.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.

3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run the FreeRTOS demo project

Import the FreeRTOS demo into DAVE

1. Start DAVE.
2. In DAVE, choose **File**, and then choose **Import**. Expand the **Infineon** folder, choose **DAVE Project**, and then choose **Next**.
3. In the **Import DAVE Projects** window, choose **Select Root Directory**, choose **Browse**, and then choose the XMC4800 demo project.

   In the directory where you unzipped your FreeRTOS download, the demo project is located in `projects/infineon/xmc4800_plus_optiga_trust_x/dave4/aws_demos/dave4`.

   Make sure that **Copy Projects Into Workspace** is cleared.
4. Choose **Finish**.

   The `aws_demos` project should be imported into your workspace and activated.
5. From the **Project** menu, choose **Build Active Project**.

   Make sure that the project builds without errors.

Run the FreeRTOS demo project

1. From the **Project** menu, choose **Rebuild Active Project** to rebuild `aws_demos` and confirm that your configuration changes are picked up.
2. From **Project Explorer**, right-click `aws_demos`, choose **Debug As**, and then choose **DAVE C/C++ Application**.
3. Double-click **GDB SEGGER J-Link Debugging** to create a debug confirmation. Choose **Debug**.
4. When the debugger stops at the breakpoint in `main()`, from the **Run** menu, choose **Resume**.

At this point, continue with the public key extraction step in Option #2: onboard private key generation. After all steps are complete, go to the AWS IoT console. The MQTT client you set up previously should display the MQTT messages sent by your device. Through the device's serial connection, you should see something like this on the UART output:

```
0 0 [Tmr Svc] Starting key provisioning...
1 1 [Tmr Svc] Write root certificate...
2 4 [Tmr Svc] Write device private key...
3 82 [Tmr Svc] Write device certificate...
4 86 [Tmr Svc] Key provisioning done...
5 291 [Tmr Svc] Wi-Fi module initialized. Connecting to AP...
.6 8046 [Tmr Svc] Wi-Fi Connected to AP. Creating tasks which use network...
7 8058 [Tmr Svc] IP Address acquired [IP Address]
8 8058 [Tmr Svc] Creating MQTT Echo Task...
9 8059 [MQTTEcho] MQTT echo attempting to connect to [MQTT Broker].
...10 23010 [MQTTEcho] MQTT echo connected.
11 23010 [MQTTEcho] MQTT echo test echoing task created.
.12 26011 [MQTTEcho] MQTT Echo demo subscribed to iotdemo/#
13 29012 [MQTTEcho] Echo successfully published 'Hello World 0'
.14 32096 [Echoing] Message returned with ACK: 'Hello World 0 ACK'
.15 37013 [MQTTEcho] Echo successfully published 'Hello World 1'
16 40080 [Echoing] Message returned with ACK: 'Hello World 1 ACK'
.17 45014 [MQTTEcho] Echo successfully published 'Hello World 2'
.18 48091 [Echoing] Message returned with ACK: 'Hello World 2 ACK'
.19 53015 [MQTTEcho] Echo successfully published 'Hello World 3'
.20 56087 [Echoing] Message returned with ACK: 'Hello World 3 ACK'
.21 61016 [MQTTEcho] Echo successfully published 'Hello World 4'
22 64083 [Echoing] Message returned with ACK: 'Hello World 4 ACK'
.23 69017 [MQTTEcho] Echo successfully published 'Hello World 5'
.24 72091 [Echoing] Message returned with ACK: 'Hello World 5 ACK'
.25 77018 [MQTTEcho] Echo successfully published 'Hello World 6'
26 80085 [Echoing] Message returned with ACK: 'Hello World 6 ACK'
.27 85019 [MQTTEcho] Echo successfully published 'Hello World 7'
.28 88086 [Echoing] Message returned with ACK: 'Hello World 7 ACK'
.29 93020 [MQTTEcho] Echo successfully published 'Hello World 8'
.30 96088 [Echoing] Message returned with ACK: 'Hello World 8 ACK'
.31 101021 [MQTTEcho] Echo successfully published 'Hello World 9'
32 104102 [Echoing] Message returned with ACK: 'Hello World 9 ACK'
.33 109022 [MQTTEcho] Echo successfully published 'Hello World 10'
.34 112047 [Echoing] Message returned with ACK: 'Hello World 10 ACK'
.35 117023 [MQTTEcho] Echo successfully published 'Hello World 11'
36 120089 [Echoing] Message returned with ACK: 'Hello World 11 ACK'
.37 122068 [MQTTEcho] MQTT echo demo finished.
```

```
38 122068 [MQTTEcho] ----Demo finished----
```

Build the FreeRTOS demo with CMake

This section covers using CMake on Windows with MingW as the native build system. For more information about using CMake with other operating systems and options, see Using CMake with FreeRTOS. (MinGW is a minimalist development environment for native Microsoft Windows applications.)

If you prefer not to use an IDE for FreeRTOS development, you can use CMake to build and run the demo applications or applications that you have developed using third-party code editors and debugging tools.

To build the FreeRTOS demo with CMake

1. Set up the GNU Arm Embedded Toolchain.

   a. Download a Windows version of the toolchain from the Arm Embedded Toolchain download page.

   > **Note**
   >
   > Due to a bug reported in the objcopy utility, we recommend that you download a version other than "8-2018-q4-major."

   b. Open the downloaded toolchain installer, and follow the instructions in the wizard.

   c. On the final page of the installation wizard, select **Add path to environment variable** to add the toolchain path to the system path environment variable.

2. Install CMake and MingW.

   For instructions, see CMake Prerequisites.

3. Create a folder to contain the generated build files (*build-folder*).

4. Change directories to your FreeRTOS download directory (*freertos*), and use the following command to generate the build files:

   ```
   cmake -DVENDOR=infineon -DBOARD=xmc4800_plus_optiga_trust_x -DCOMPILER=arm-gcc -S .
    -B build-folder -G "MinGW Makefiles" -DAFR_ENABLE_TESTS=0
   ```

5. Change directories to the build directory (*build-folder*), and use the following command to build the binary:

   ```
   cmake --build . --parallel 8
   ```

   This command builds the output binary `aws_demos.hex` to the build directory.

6. Flash and run the image with JLINK.

   a. From the build directory (*build-folder*), use the following commands to create a flash script:

   ```
   echo loadfile aws_demos.hex > flash.jlink
   echo r >> flash.jlink
   echo g >> flash.jlink
   echo q >> flash.jlink
   ```

   b. Flash the image using the JLNIK executable.

   ```
   JLINK_PATH\JLink.exe  -device XMC4800-2048 -if SWD -speed auto -CommanderScript
    flash.jlink
   ```

   The application logs should be visible through the serial connection that you established with the board. Continue to the public key extraction step in Option #2: onboard private key generation. After all the steps are complete, go to the AWS IoT console. The MQTT client you set up previously should display the MQTT messages sent by your device.

Troubleshooting

For general troubleshooting information, see Troubleshooting getting started.

Getting started with the MW32x AWS IoT Starter Kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The AWS IoT Starter Kit is a development kit based on the 88MW320/88MW322, the latest integrated Cortex M4 Microcontroller from NXP, that integrates 802.11b/g/n Wi-Fi on a single microcontroller chip. The development kit is FCC certified. For more information, see the AWS Partner Device Catalog to purchase one from our partner. The 88MW320/88MW322 modules are also FCC certified and available for customization and volume sale.

This getting started guide shows you how to cross compile your application together with the SDK on a host computer, then load the generated binary file onto the board using the tools provided with the SDK. When the application starts running on the board, you can debug or interact with it from the Serial console on your host computer.

Ubuntu 16.04 is the host platform supported for development and debugging. You might be able to use other platforms, but these are not officially supported. You must have permissions to install software on the host platform. The following external tools required to build the SDK:

- Ubuntu 16.04 host platform
- ARM toolchain version 4_9_2015q3
- Eclipse 4.9.0 IDE

The ARM toolchain is required to cross compile your application and the SDK. The SDK takes advantage of the latest versions of the toolchain to optimize the image footprint and fit more functionality into less space. This guide assumes that you're using version 4_9_2015q3 of the toolchain. Using older versions of the toolchain isn't recommended. The development kit is pre-flashed with Wireless Microcontroller Demo project firmware.

Topics

- Setting up your hardware
- Setting up the development environment
- Build and run the FreeRTOS demo project
- Debugging
- Troubleshooting

Setting up your hardware

Connect the MW32x board to your laptop by using a mini-USB to USB cable. Connect the mini-USB cable to the only mini-USB connector present on the board. You don't need a jumper change.

If the board is connected to a laptop or desktop computer, you don't need an external power supply.

This USB connection provides the following:

- Console access to the board. A virtual tty/com port is registered with the development host that can be used to access the console.
- JTAG access to the board. This can be used to load or unload firmware images into the RAM or flash of the board, or for debugging purposes.

Setting up the development environment

For development purposes, the minimum requirement is the ARM toolchain and the tools bundled with the SDK. The following sections provide details on the ARM toolchain setup.

GNU Toolchain

The SDK officially supports the GCC Compiler toolchain. The cross-compiler toolchain for GNU ARM is available at GNU Arm Embedded Toolchain 4.9-2015-q3-update.

The build system is configured to use the GNU toolchain by default. The Makefiles assume that the GNU compiler toolchain binaries are available on the user's PATH and can be invoked from the Makefiles. The Makefiles also assume that the file names of the GNU toolchain binaries are prefixed with `arm-none-eabi-`.

The GCC toolchain can be used with GDB to debug with OpenOCD (bundled with the SDK). This provides the software that interfaces with JTAG.

We recommend version 4_9_2015q3 of the gcc-arm-embedded toolchain.

Linux Toolchain Setup Procedure

Follow these steps to set up the GCC toolchain in Linux.

1. Download the toolchain tarball available at GNU Arm Embedded Toolchain 4.9-2015-q3-update. The file is `gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2`.
2. Copy the file to a directory of your choice. Make sure there are no spaces in the directory name.
3. Use the following command to untar the file.

```
tar -vxf filename
```

4. Add the path of the installed toolchain to the system PATH. For example, append the following line at the end of the `.profile` file located in `/home/user-name` directory.

```
PATH=$PATH:path to gcc-arm-none-eabit-4_9_2015_q3/bin
```

> **Note**
>
> Newer distributions of Ubuntu might come with a Debian version of the GCC Cross Compiler. If so, you must remove the native Cross Compiler and follow the above setup procedure.

Working with a Linux development host

Any modern Linux desktop distribution such as Ubuntu or Fedora can be used. However, we recommend that you upgrade to the most recent release. The following steps have been verified to work on Ubuntu 16.04 and assume that you're using that version.

Installing Packages

The SDK includes a script to enable quick setup of your development environment on a newly setup Linux machine. The script attempts to auto detect the machine type and install the appropriate software, including C libraries, USB library, FTDI library, ncurses, python, and latex. In this section, the generic directory name *amzsdk_bundle-x.y.z* indicates the AWS SDK root directory. The actual directory name might be different. You must have root privileges.

- Navigate to the *amzsdk_bundle-x.y.z/* directory and run this command.

```
./lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/bin/installpkgs.sh
```

Avoiding sudo

In this guide, the `flashprog` operation uses the `flashprog.py` script to flash the NAND of the board, as explained below. Similarly, the `ramload` operation uses the `ramload.py` script to copy

the firmware image from the host directly to the RAM of the microcontroller, without flashing the NAND.

You can configure your Linux development host to perform the `flashprog` and `ramload` operations without requiring the `sudo` command each time. To do this, run the following command.

```
./lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/bin/perm_fix.sh
```

> **Note**
>
> You must configure your Linux development host permissions in this way to ensure a smooth Eclipse IDE experience.

Setting up the Serial Console

Insert the USB cable into the Linux host USB slot. This triggers the detection of the device. You should see messages like the following in the `/var/log/messages` file, or after executing the `dmesg` command.

```
Jan 6 20:00:51 localhost kernel: usb 4-2: new full speed USB device using uhci_hcd and
 address 127
Jan 6 20:00:51 localhost kernel: usb 4-2: configuration #1 chosen from 1 choice
Jan 6 20:00:51 localhost kernel: ftdi_sio 4-2:1.0: FTDI USB Serial Device converter
 detected
Jan 6 20:00:51 localhost kernel: ftdi_sio: Detected FT2232C
Jan 6 20:00:51 localhost kernel: usb 4-2: FTDI USB Serial Device converter now attached
 to ttyUSB0
Jan 6 20:00:51 localhost kernel: ftdi_sio 4-2:1.1: FTDI USB Serial Device converter
 detected
Jan 6 20:00:51 localhost kernel: ftdi_sio: Detected FT2232C
Jan 6 20:00:51 localhost kernel: usb 4-2: FTDI USB Serial Device converter now attached
 to ttyUSB1
```

Verify that two ttyUSB devices have been created. The second ttyUSB is the serial console. In the example above, this is named "ttyUSB1".

In this guide, we use minicom to see the serial console output. You might also use other serial programs such as `putty`. Run the following command to execute minicom in setup mode.

```
minicom -s
```

In minicom, navigate to **Serial Port Setup** and capture the following settings.

```
| A - Serial Device : /dev/ttyUSB1
| B - Lockfile Location : /var/lock
| C - Callin Program :
| D - Callout Program :
| E - Bps/Par/Bits : 115200 8N1
| F - Hardware Flow Control : No
| G - Software Flow Control : No
```

You can save these settings in minicom for future use. The minicom window now shows messages from the serial console.

Choose the serial console window and press the **Enter** key. This displays a hash (#) on the screen.

> **Note**
>
> The development boards include an FTDI silicon device. The FTDI device exposes two USB interfaces for the host. The first interface is associated with the JTAG functionality of the MCU and the second interface is associated with the physical UARTx port of the MCU.

Installing OpenOCD

OpenOCD is software that provides debugging, in-system programming, and boundary-scan testing for embedded target devices.

OpenOCD version 0.9 is required. It's also required for Eclipse functionality. If an earlier version, such as version 0.7, was installed on your Linux host, remove that repository with the appropriate command for the Linux distribution that you're currently using.

Run the standard Linux command to install OpenOCD,

```
apt-get install openocd
```

If the above command doesn't install version 0.9 or later, use the following procedure to download and compile the openocd source code.

To install OpenOCD

1. Run the following command to install libusb-1.0.

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

2. Download the openocd 0.9.0 source code from http://openocd.org/.
3. Extract openocd and navigate to the directory where you extracted it.
4. Configure openocd with the following command.

```
./configure --enable-ftdi --enable-jlink
```

5. Run the make utility to compile opencd.

```
make install
```

Setting up Eclipse

> **Note**
>
> This section assumes that you have completed the steps in Avoiding sudo

Eclipse is the preferred IDE for application development and debugging. It provides a rich, user-friendly IDE with integrated debugging support, including thread aware debugging. This section describes the common Eclipse setup for all the development hosts that are supported.

To set up Eclipse

1. Download and install the Java Run Time Environment (JRE).

   Eclipse requires that you install the JRE. We recommend that you install this first, although it can be installed after you install Eclipse. The JRE version (32/64 bit) must match the version of Eclipse (32/64 bit). You can download the JRE from Java SE Runtime Environment 8 Downloads on the Oracle website.
2. Download and install the "Eclipse IDE for C/C++ Developers" from http://www.eclipse.org. Eclipse version 4.9.0 or later is supported. The installation only requires you to extract the downloaded archive. You run the platform-specific Eclipse executable to start the application.

Build and run the FreeRTOS demo project

There are two ways to run the FreeRTOS demo project:

- Use the command line.
- Use the Eclipse IDE.

This topic covers both options.

Provisioning

- Depending on whether you use the test or demo application, set the provisioning data in one of the following files:

  - `./tests/common/include/aws_clientcredential.h`
  - `./demos/common/include/aws_clientcredential.h`

  For example:

```
#define clientcredentialWIFI_SSID "Wi-Fi SSID"
#define clientcredentialWIFI_PASSWORD "Wi-Fi password"
#define clientcredentialWIFI_SECURITY "Wi-Fi security"
```

  Note

  You can enter the following Wi-Fi security values:

  - `eWiFiSecurityOpen`
  - `eWiFiSecurityWEP`
  - `eWiFiSecurityWPA`
  - `eWiFiSecurityWPA2`

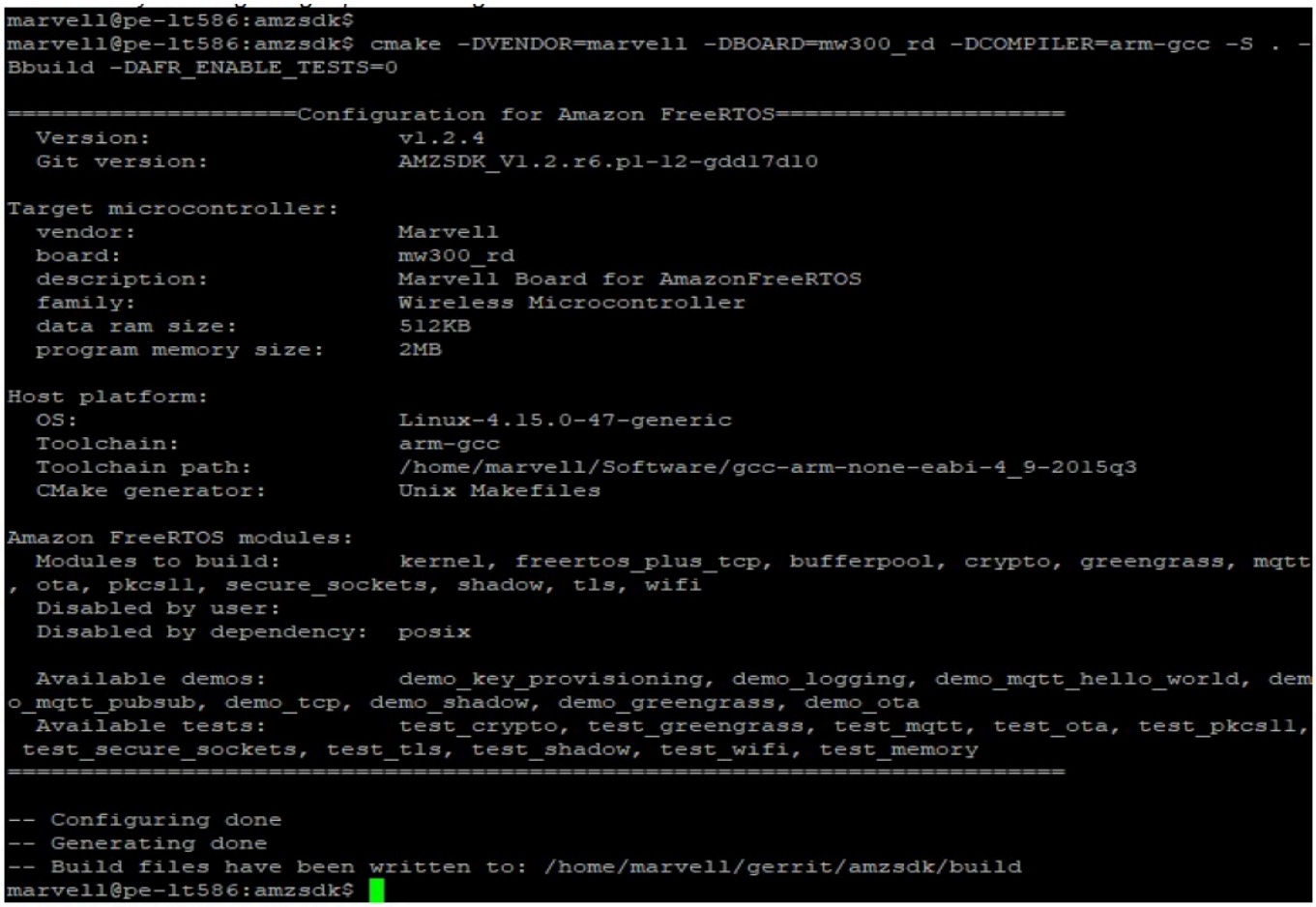
  The SSID and password should be enclosed in double quotes.

Build and run the FreeRTOS demo using the command line

1. Use the following command to start building the demo application.

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
DAFR_ENABLE_TESTS=0
```

   Make sure you get the same output as shown in the following example.



2. Navigate to the build directory.

```
cd build
```

3. Run the make utility to build the application.

```
make all -j4
```

Make sure you get the same output as shown in the following figure:



4. Use the following commands to build a test application.

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
DAFR_ENABLE_TESTS=1
cd build
make all -j4
```

   Run the `cmake` command every time you switch between the `aws_demos project` and the `aws_tests project`.

5. Write the firmware image to the flash of the development board. The firmware will execute after the development board is reset. You must build the SDK before you flash the image to the microcontroller.

   a. Before you flash the firmware image, prepare the development board's flash with the common components Layout and Boot2. Use the following commands.

   ```
   cd amzsdk_bundle-x.y.z
   ./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py -l ./vendors/
   marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt --boot2 ./vendors/
   marvell/WMSDK/mw320/boot2/bin/boot2.bin
   ```

   The `flashprog` command initiates the following:

- Layout – The flashprog utility is first instructed to write a layout to the flash. The layout is similar to partition information for the flash. The default layout is located at `/lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt`.
- Boot2 – This is the boot-loader used by the WMSDK. The `flashprog` command also writes a bootloader to the flash. It's the bootloader's job to load the microcontroller's firmware image after it's flashed. Make sure you get the same output as shown in the figure below.

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245458s (115.728 KiB/s)
verified 29088 bytes in 0.350004s (81.160 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Erasing primary flash...done
Writing new flash layout...done
Writing "boot2" @0x0 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

b. The firmware uses the Wi-Fi chipset for its functionality, and the Wi-Fi chipset has its own firmware that must also be present in the flash. You use the `flashprog.py` utility to flash the Wi-Fi firmware in the same way that you did to flash the Boot2 boot-loader and the MCU firmware. Use the following commands to flash the Wi-Fi firmware.

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --wififw ./
vendors/marvell/WMSDK/mw320/wifi-firmware/mw30x/mw30x_uapsta_W14.88.36.p135.bin
```

Make sure the output of the command is similar to the figure below.

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245498s (115.709 KiB/s)
verified 29088 bytes in 0.350229s (81.108 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "wififw" @0x12a000 (primary).....done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

c. Use the following commands to flash the MCU firmware.

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_demos.bin -r
```

d. Reset the board. You should see the logs for the demo app.

e. To run the test app, flash the `aws_tests.bin` binary located at the same directory.

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_tests.bin -r
```

Your command output should be similar to the one shown in the figure below.

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245499s (115.708 KiB/s)
verified 29088 bytes in 0.350231s (81.107 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "mcufw" @0x6a000 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
Resetting board...
Using OpenOCD interface file ftdi.cfg
Open On-Chip Debugger 0.9.0 (2015-07-15-15:28)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 3000 kHz
adapter_nsrst_delay: 100
Info : auto-selecting first available session transport "jtag". To override use 'transport
 select <transport>'.
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
sh_load
Info : clock speed 3000 kHz
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
Info : wmcore.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
shutdown command invoked
Resetting board done...
```

6. After you flash the firmware and reset the board, the demo app should start as shown in the figure below.

```
Network connection successful.

Wi-Fi Connected to AP. Creating tasks which use network...
2 6293 [Startup Hook] Write certificate...
3 6296 [Startup Hook] Write device private key...
4 6362 [Startup Hook] Creating MQTT Echo Task...
6 11668 [MQTTEcho] MQTT echo connected.to connect to a2wtm15blvjjs8-ats.iot.us-east-2.amazonaws.com
7 11668 [MQTTEcho] MQTT echo test echoing task created.
8 11961 [MQTTEcho] MQTT Echo demo subscribed to freertos/demos/echo
9 12248 [MQTTEcho] Echo successfully published 'Hello World 0'
10 12591 [Echoing] Message returned with ACK: 'Hello World 0 ACK'
11 17633 [MQTTEcho] Echo successfully published 'Hello World 1'
12 17927 [Echoing] Message returned with ACK: 'Hello World 1 ACK'
13 22953 [MQTTEcho] Echo successfully published 'Hello World 2'
14 23276 [Echoing] Message returned with ACK: 'Hello World 2 ACK'
15 28245 [MQTTEcho] Echo successfully published 'Hello World 3'
16 28575 [Echoing] Message returned with ACK: 'Hello World 3 ACK'
17 33542 [MQTTEcho] Echo successfully published 'Hello World 4'
18 33980 [Echoing] Message returned with ACK: 'Hello World 4 ACK'
19 38823 [MQTTEcho] Echo successfully published 'Hello World 5'
20 39279 [Echoing] Message returned with ACK: 'Hello World 5 ACK'
21 44139 [MQTTEcho] Echo successfully published 'Hello World 6'
22 44501 [Echoing] Message returned with ACK: 'Hello World 6 ACK'
23 49516 [MQTTEcho] Echo successfully published 'Hello World 7'
24 50270 [Echoing] Message returned with ACK: 'Hello World 7 ACK'
25 54796 [MQTTEcho] Echo successfully published 'Hello World 8'
26 55129 [Echoing] Message returned with ACK: 'Hello World 8 ACK'
27 60080 [MQTTEcho] Echo successfully published 'Hello World 9'
28 60389 [Echoing] Message returned with ACK: 'Hello World 9 ACK'
29 65378 [MQTTEcho] Echo successfully published 'Hello World 10'
30 65998 [Echoing] Message returned with ACK: 'Hello World 10 ACK'
31 70658 [MQTTEcho] Echo successfully published 'Hello World 11'
32 70964 [Echoing] Message returned with ACK: 'Hello World 11 ACK'
33 75958 [MQTTEcho] MQTT echo demo finished.
34 75958 [MQTTEcho] ----Demo finished----
```

7. (Optional) As an alternative method to test your image, use the flashprog utility to copy the microcontroller image from the host directly into the microcontroller RAM. The image isn't copied in the flash, so it will be lost after you reboot the microcontroller.

   Loading the firmware image into the SRAM is a faster operation because it launches the execution file immediately. This method is used primarily for iterative development.

   Use the following commands to load the firmware into the SRAM.

   ```
   cd amzsdk_bundle-x.y.z
   ./lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/OpenOCD/ramload.py
    build/cmake/vendors/marvell/mw300_rd/aws_demos.axf
   ```

   The command output is shown in the figure below.

```
Using OpenOCD interface file ftdi.cfg
Open On-Chip Debugger 0.9.0 (2015-07-15-15:28)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 3000 kHz
adapter_nsrst_delay: 100
Info : auto-selecting first available session transport "jtag". To override use 'transport
 select <transport>'.
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
sh_load
Info : clock speed 3000 kHz
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
Info : wmcore.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
75072 bytes written at address 0x00100000
8 bytes written at address 0x00112540
468 bytes written at address 0x20000040
downloaded 75548 bytes in 0.636127s (115.979 KiB/s)
verified 75548 bytes in 0.959023s (76.930 KiB/s)
shutdown command invoked
```

When the command execution is complete, you should see the logs of the demo app.

Build and run the FreeRTOS demo using the Eclipse IDE

1. Before you set up an Eclipse workspace, you must run the `cmake` command.

   Run the following command to work with the `aws_demos` Eclipse project.

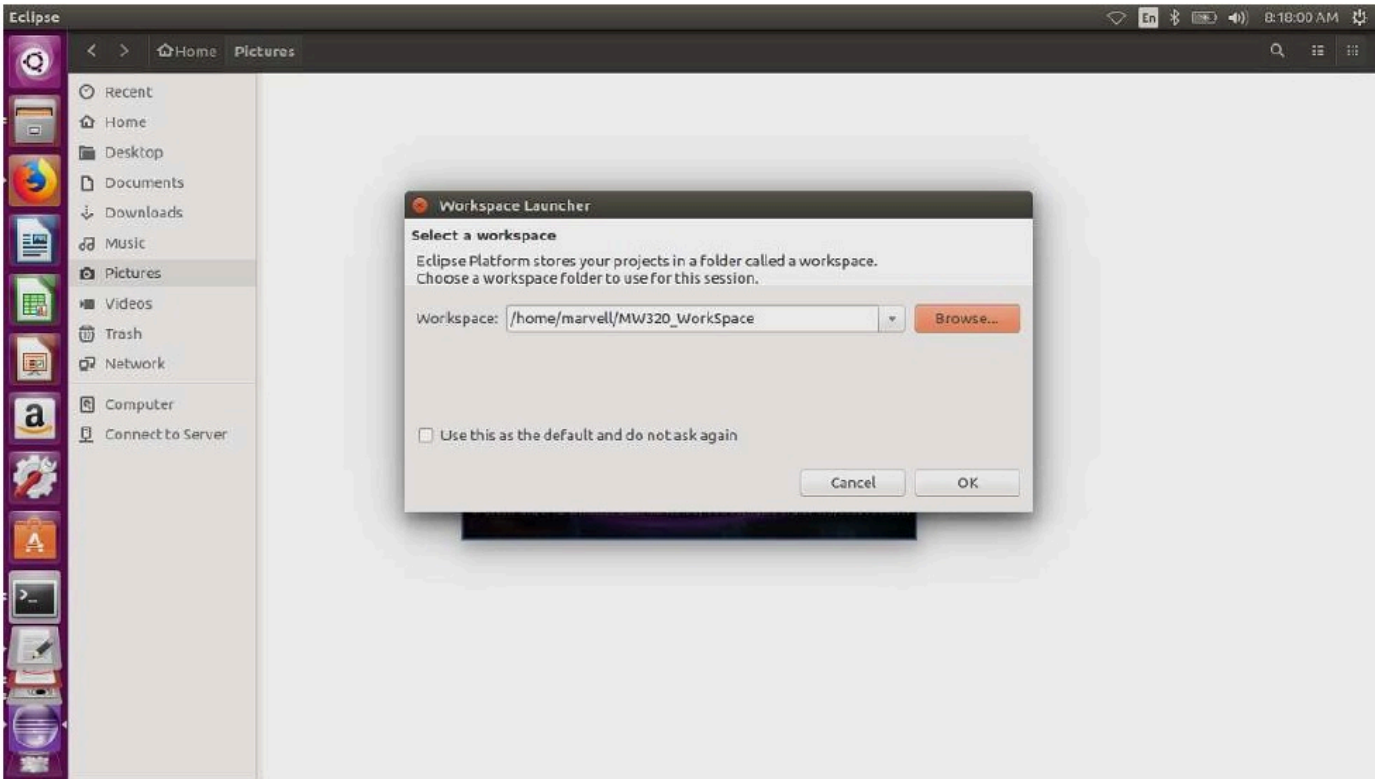
   ```
   cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
   DAFR_ENABLE_TESTS=0
   ```

   Run the following command to work with the `aws_tests` Eclipse project.

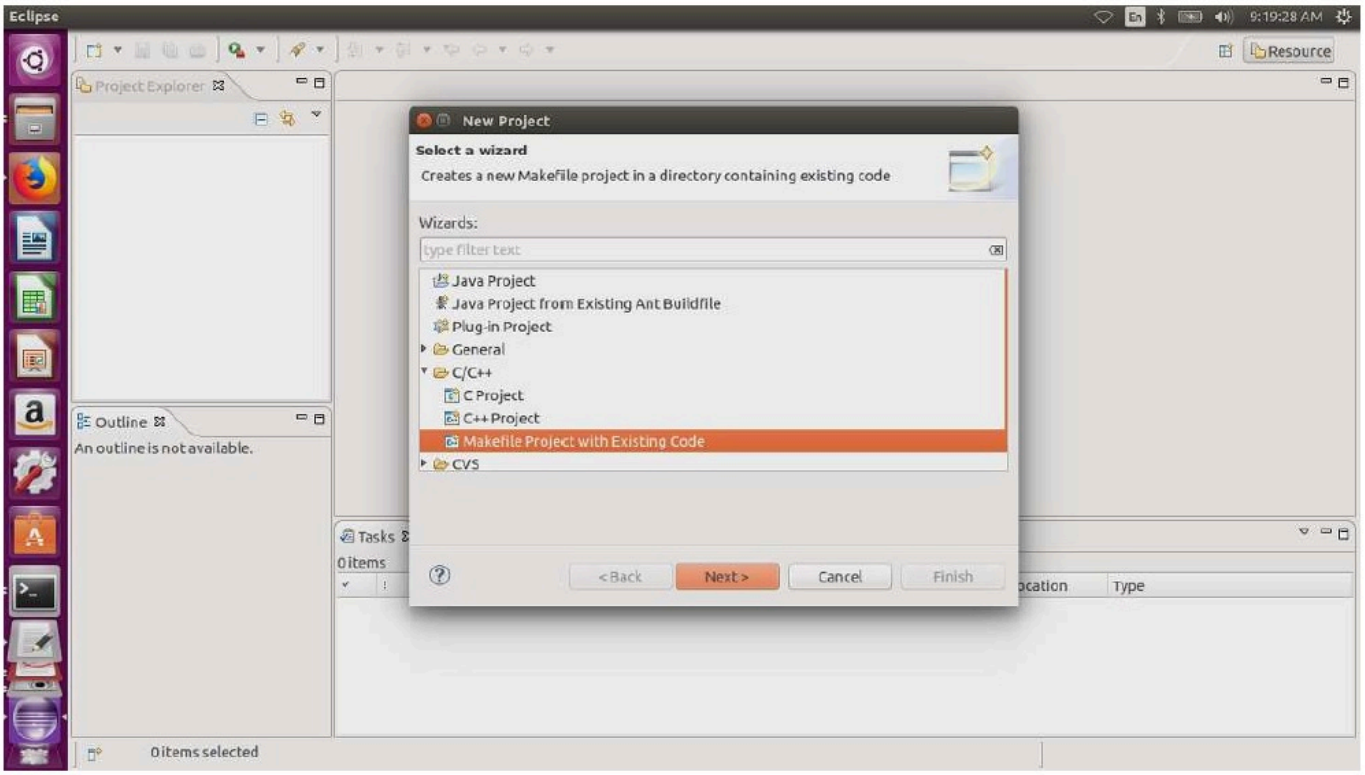
   ```
   cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
   DAFR_ENABLE_TESTS=1
   ```

> **Tip**
>
> Run the `cmake` command every time you switch between the `aws_demos` project and the `aws_tests` project.

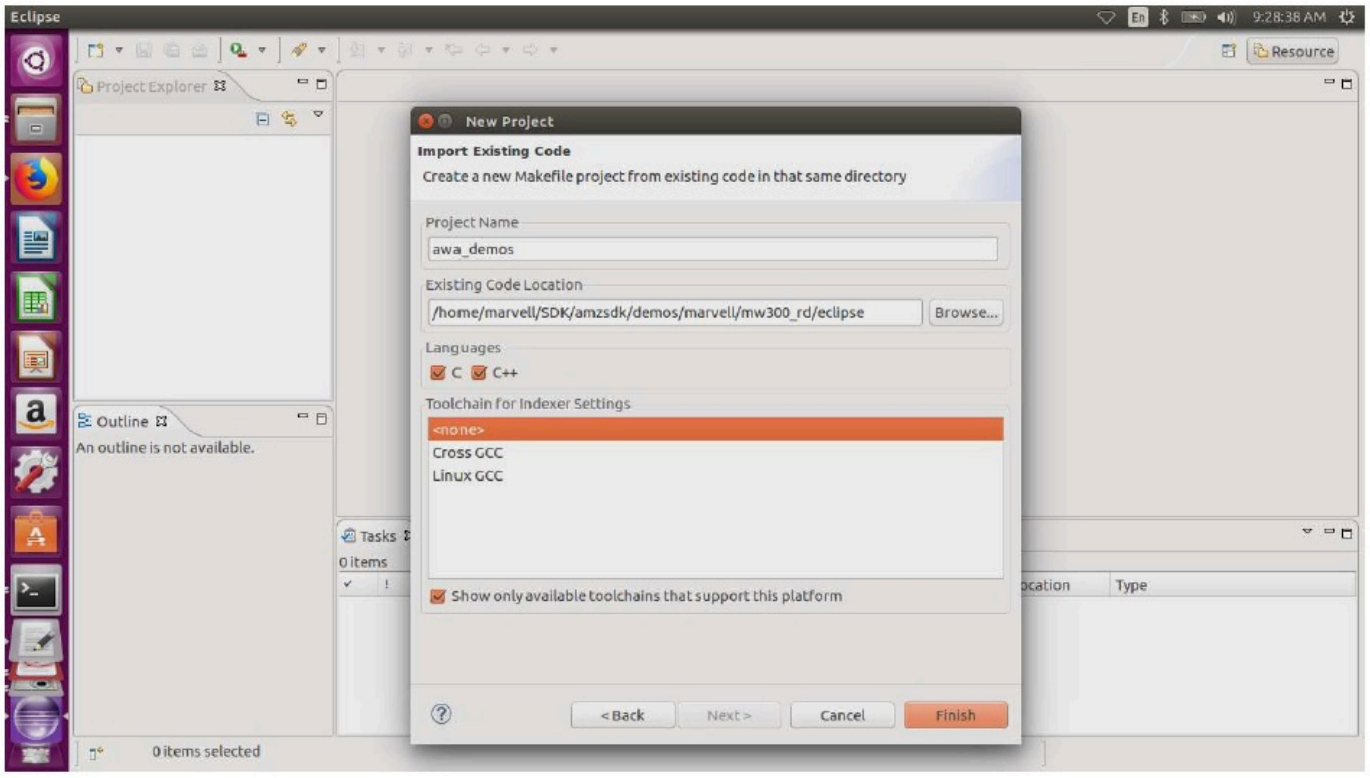
2. Open Eclipse and, when prompted, choose your Eclipse workspace as shown in the figure below.



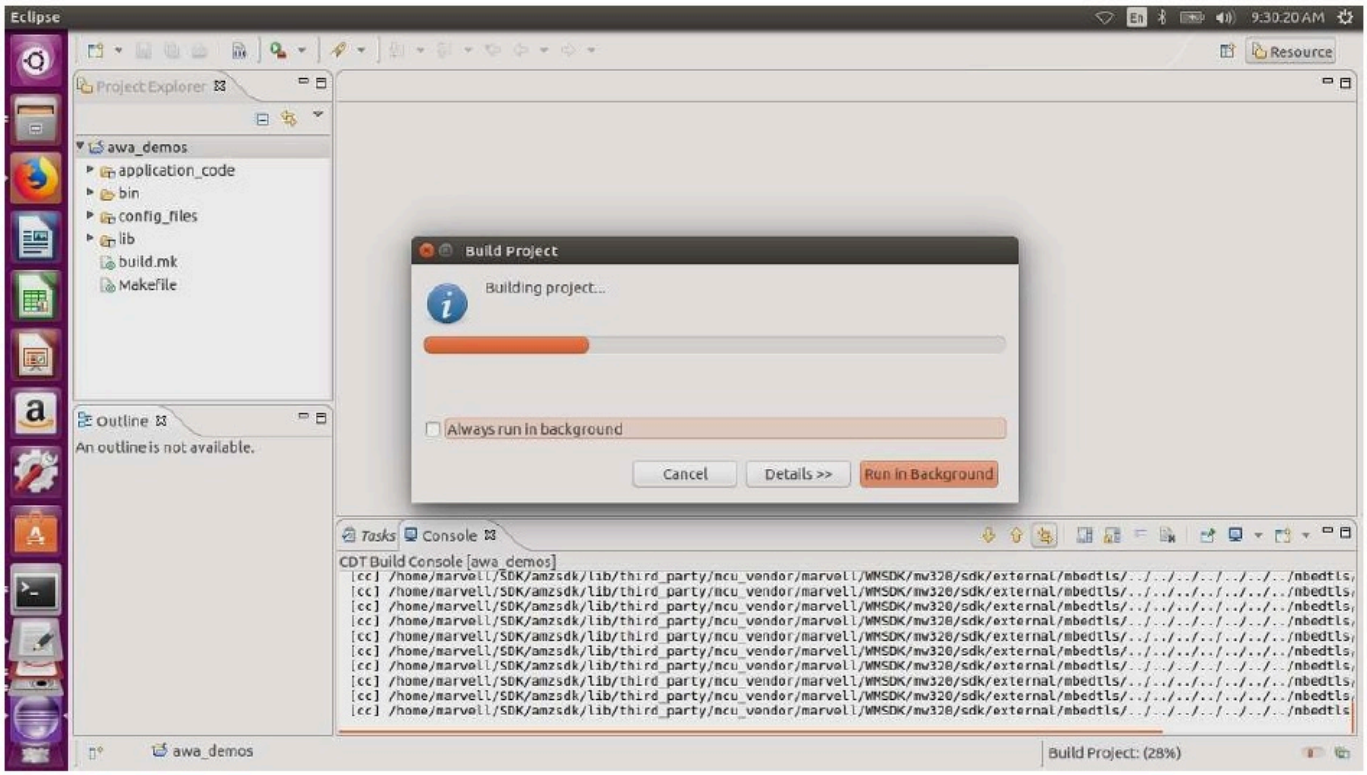
3. Choose the option to create a **Makefile Project: with Existing Code** as shown in the figure below.



4. Choose **Browse**, specify the directory of the existing code, and then choose **Finish**.



5. In the navigation pane, choose **aws_demos** in the project explorer. Right-click **aws_demos** to open the menu, then choose **Build**.



If the build succeeds, it generates the `build/cmake/vendors/marvell/mw300_rd/aws_demos.bin` file.

6. Use the command line tools to flash the Layout file (`layout.txt`), the Boot2 binary (`boot2.bin`), the MCU firmware binary (`aws_demos.bin`), and the Wi-Fi firmware.

   a. Before you flash the firmware image, prepare the development board's flash with the common components, Layout and Boot2. Use the following commands.

   ```
   cd amzsdk_bundle-x.y.z
   ./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py -l ./vendors/
   marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt --boot2 ./vendors/
   marvell/WMSDK/mw320/boot2/bin/boot2.bin
   ```

   The `flashprog` command initiates the following:

   - Layout – The flashprog utility is first instructed to write a layout to the flash. The layout is similar to partition information for the flash. The default layout is located at `/lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt`.

- Boot2 – This is the boot-loader used by the WMSDK. The flashprog command also writes a bootloader to the flash. It is the bootloader's job to load the microcontroller's firmware image after it is flashed. Make sure you get the same output as shown in the figure below.

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245458s (115.728 KiB/s)
verified 29088 bytes in 0.350004s (81.160 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Erasing primary flash...done
Writing new flash layout...done
Writing "boot2" @0x0 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

b. The firmware uses the Wi-Fi chipset for its functionality, and the Wi-Fi chipset has its own firmware that must also be present in the flash. You use the `flashprog.py` utility to flash the Wi-Fi firmware in the same way that you did to flash the boot2 boot-loader and the MCU firmware. Use the following commands to flash the Wi-Fi firmware.

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --wififw ./
vendors/marvell/WMSDK/mw320/wifi-firmware/mw30x/mw30x_uapsta_W14.88.36.p135.bin
```

Make sure the output of the command is similar to the figure below.

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245498s (115.709 KiB/s)
verified 29088 bytes in 0.350229s (81.108 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "wififw" @0x12a000 (primary).....done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

c. Use the following commands to flash the MCU firmware.

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_demos.bin -r
```

d. Reset the board. You should see the logs for the demo app.

e. To run the test app, flash the `aws_tests.bin` binary located at the same directory.

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_tests.bin -r
```

Your command output should be similar to the one shown in the figure below.

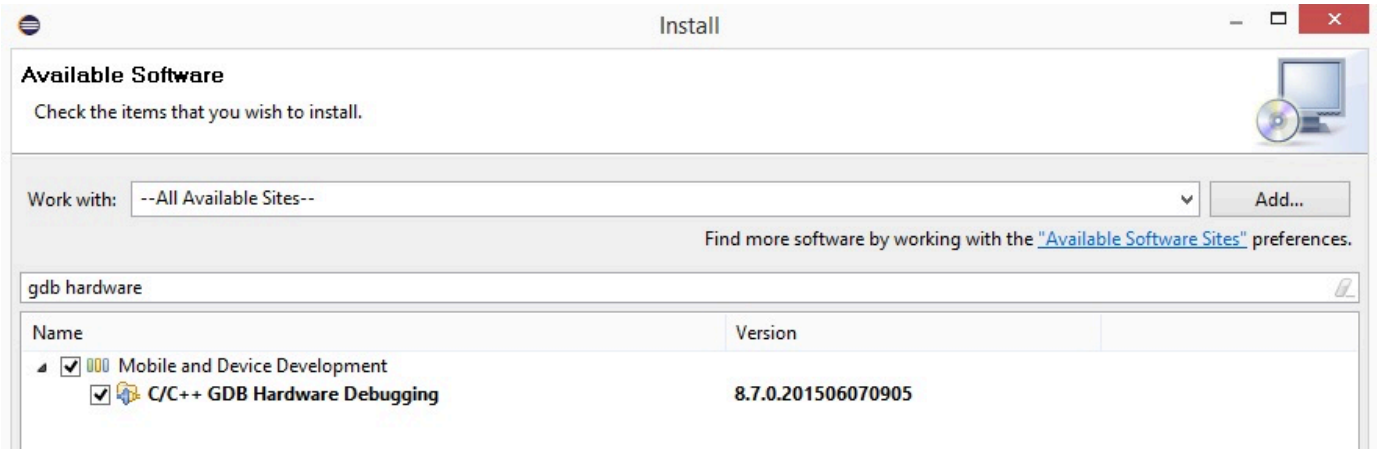
```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245499s (115.708 KiB/s)
verified 29088 bytes in 0.350231s (81.107 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "mcufw" @0x6a000 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
Resetting board...
Using OpenOCD interface file ftdi.cfg
Open On-Chip Debugger 0.9.0 (2015-07-15-15:28)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 3000 kHz
adapter_nsrst_delay: 100
Info : auto-selecting first available session transport "jtag". To override use 'transport
 select <transport>'.
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
sh_load
Info : clock speed 3000 kHz
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
Info : wmcore.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
shutdown command invoked
Resetting board done...
```

Debugging

- Start Eclipse and choose **Help** and then choose **Install new software**. In the **Work with** menu, choose **All Available Sites**. Enter the filter text `GDB Hardware`. Select the **C/C++ GDB Hardware Debugging** option and install the plugin.



Troubleshooting

Network issues

Check your network credentials. See "Provisioning" in Build and run the FreeRTOS demo project.

Enabling additional logs

- Enable board specific logs.

  Enable calls to `wmstdio_init(UART0_ID, 0)` in the function `prvMiscInitialization` in the `main.c` file for tests or demos.

- Enabling Wi-Fi logs

  Enable the macro `CONFIG_WLCMGR_DEBUG` in the *`freertos`*`/vendors/marvell/WMSDK/mw320/sdk/src/incl/autoconf.h` file.

Using GDB

We recommend that you use the `arm-none-eabi-gdb` and gdb command files packaged along with the SDK. Navigate to the directory.

```
cd freertos/lib/third_party/mcu_vendor/marvell/WMSDK/mw320
```

Run the following command (on a single line) to connect to GDB.

```
arm-none-eabi-gdb -x ./sdk/tools/OpenOCD/gdbinit ../../../../../../build/cmake/
vendors/marvell/mw300 _rd/aws_demos.axf
```

Getting started with the MediaTek MT7697Hx development kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the MediaTek MT7697Hx Development Kit. If you do not have the MediaTek MT7697Hx Development Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains instructions for the following getting started steps:

1. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
2. Cross compiling a FreeRTOS demo application to a binary image.
3. Loading the application binary image to your board, and then running the application.
4. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Set up your development environment

Before you set up your environment, connect your computer to the USB port on the MediaTek MT7697Hx Development Kit.

Download and install Keil MDK

You can use the GUI-based Keil Microcontroller Development Kit (MDK) to configure, build, and run FreeRTOS projects on your board. Keil MDK includes the µVision IDE and the µVision Debugger.

> **Note**
>
> Keil MDK is supported on Windows 7, Windows 8, and Windows 10 64-bit machines only.

To download and install Keil MDK

1. Go to the Keil MDK Getting Started page, and choose **Download MDK-Core**.
2. Enter and submit your information to register with Keil.
3. Right-click the MDK executable and save the Keil MDK installer to your computer.
4. Open the Keil MDK installer and follow the steps to completion. Make sure that you install the MediaTek device pack (MT76x7 Series).

Establish a serial connection

Connect the board to your host computer with a USB cable. A COM port appears in the Windows Device Manager. For debugging, you can open a session to the port with a terminal utility tool such as HyperTerminal or TeraTerm.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run the FreeRTOS demo project with Keil MDK

To build the FreeRTOS demo project in Keil µVision

1. From the **Start** menu, open Keil µVision 5.
2. Open the `projects/mediatek/mt7697hx-dev-kit/uvision/aws_demos/aws_demos.uvprojx` project file.
3. From the menu, choose **Project**, and then choose **Build target**.

   After the code is built, you see the demo executable file at `projects/mediatek/mt7697hx-dev-kit/uvision/aws_demos/out/Objects/aws_demo.axf`.

To run the FreeRTOS demo project

1. Set the MediaTek MT7697Hx Development Kit to PROGRAM mode.

   To set the kit to PROGRAM mode, press and hold the **PROG** button. With the **PROG** button still pressed, press and release the **RESET** button, and then release the **PROG** button.
2. From the menu, choose **Flash**, and then choose **Configure Flash Tools**.
3. In **Options for Target 'aws_demo'**, choose the **Debug** tab. Select **Use**, set the debugger to **CMSIS-DAP Debugger**, and then choose **OK**.
4. From the menu, choose **Flash**, and then choose **Download**.

   µVision notifies you when the download is complete.
5. Use a terminal utility to open the serial console window. Set the serial port to 115200 bps, none-parity, 8-bits, and 1 stop-bit.
6. Choose the **RESET** button on your MediaTek MT7697Hx Development Kit.

Troubleshooting

Debugging FreeRTOS projects in Keil µVision

Currently, you must edit the MediaTek package that is included with Keil µVision before you can debug the FreeRTOS demo project for MediaTek with Keil µVision.

To edit the MediaTek package for debugging FreeRTOS projects

1. Find and open the `Keil_v5\ARM\PACK\.Web\MediaTek.MTx.pdsc` file in your Keil MDK installation folder.
2. Replace all instances of `flag = Read32(0x20000000);` with `flag = Read32(0x0010FBFC);`.
3. Replace all instances of `Write32(0x20000000, 0x76877697);` with `Write32(0x0010FBFC, 0x76877697);`.

To start debugging the project

1. From the menu, choose **Flash**, and then choose **Configure Flash Tools**.
2. Choose the **Target** tab, and then choose **Read/Write Memory Areas**. Confirm that IRAM1 and IRAM2 are both selected.
3. Choose the **Debug** tab, and then choose **CMSIS-DAP Debugger**.
4. Open `vendors/mediatek/boards/mt7697hx-dev-kit/aws_demos/application_code/main.c`, and set the macro `MTK_DEBUGGER` to 1.
5. Rebuild the demo project in µVision.
6. Set the MediaTek MT7697Hx Development Kit to PROGRAM mode.

   To set the kit to PROGRAM mode, press and hold the **PROG** button. With the **PROG** button still pressed, press and release the **RESET** button, and then release the **PROG** button.
7. From the menu, choose **Flash**, and then choose **Download**.

   µVision notifies you when the download is complete.
8. Press the **RESET** button on your MediaTek MT7697Hx Development Kit.
9. From the µVision menu, choose **Debug**, and then choose **Start/Stop Debug Session**. The **Call Stack + Locals** window opens when you start the debug session.
10. From the menu, choose **Debug**, and then choose **Stop** to pause the code execution. The program counter stops at the following line:

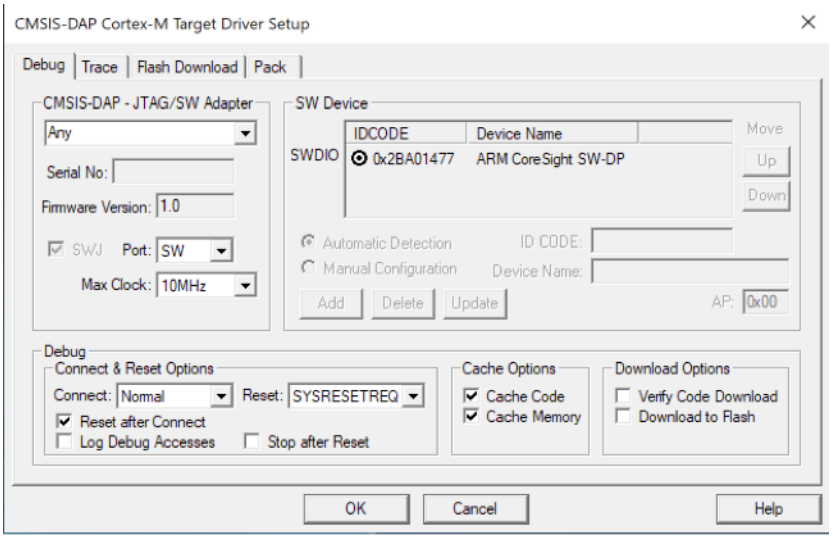
    ```
    { volatile int wait_ice = 1 ; while ( wait_ice ) ; }
    ```
11. In the **Call Stack + Locals** window, change the value for `wait_ice` to `0`.
12. Set breakpoints in your project's source code, and run the code.

Troubleshooting the IDE debugger settings

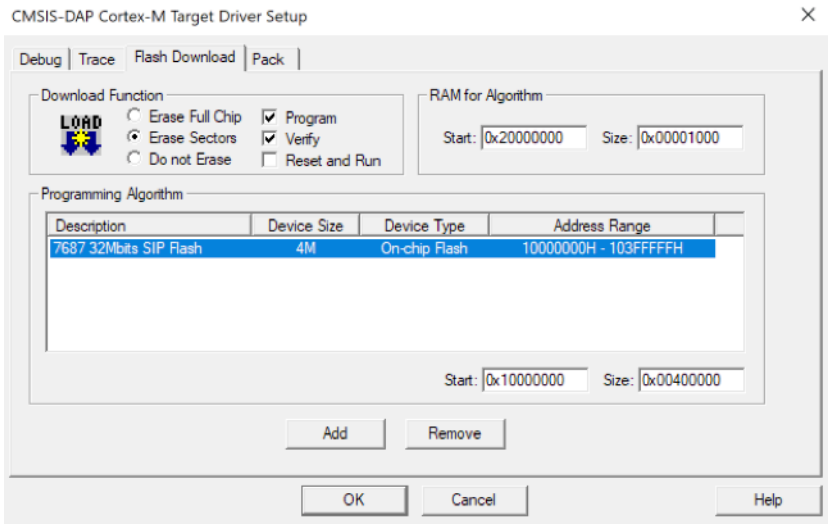
If you are having trouble debugging an application, your debugger settings might be incorrect.

To verify that your debugger settings are correct

1. Open Keil µVision.
2. Right-click the `aws_demos` project, choose **Options**, and under the **Utilities** tab, choose **Settings**, next to **"-- Use Debug Driver --"**.
3. Verify that the settings under the **Debug** tab appear as follows:



4. Verify that the settings under the **Flash Download** tab appear as follows:



For general troubleshooting information about Getting Started with FreeRTOS, see [Troubleshooting getting started].

Getting started with the Microchip Curiosity PIC32MZ EF

Important

This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Note

In agreement with Microchip, we are removing the Curiosity PIC32MZEF (DM320104) from the FreeRTOS Reference Integration repository main branch and will no longer carry it in new releases. Microchip has issued an official notice that the PIC32MZEF (DM320104) is no longer recommended for new designs. The PIC32MZEF projects and source code can still be accessed through the previous release tags. Microchip recommends that customers use the Curiosity PIC32MZ-EF-2.0 Development board (DM320209) for new designs. The PIC32MZv1 platform can still be found in v202012.00 of the FreeRTOS Reference Integration repository. However, the platform is no longer supported by v202107.00 of the FreeRTOS Reference.

This tutorial provides instructions for getting started with the Microchip Curiosity PIC32MZ EF. If you do not have the Microchip Curiosity PIC32MZ EF bundle, visit the AWS Partner Device Catalog to purchase one from our partner.

The bundle includes the following items:

- Curiosity PIC32MZ EF Development Board
- MikroElectronika USB UART click Board
- MikroElectronika WiFi 7 click Board
- PIC32 LAN8720 PHY daughter board

You also need the following items for debugging:

- MPLAB Snap In-Circuit Debugger

- (Optional) PICkit 3 Programming Cable Kit

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions.

⚠ Important

- In this topic, the path to the FreeRTOS download directory is referred to as *`freertos`*.
- Space characters in the *`freertos`* path can cause build failures. When you clone or copy the repository, make sure the path that you create doesn't contain space characters.
- The maximum length of a file path on Microsoft Windows is 260 characters. Long FreeRTOS download directory paths can cause build failures.
- Because the source code may contain symbolic links, if you're using Windows to extract the archive, you may have to:
  - Enable Developer Mode or,
  - Use a console that is elevated as administrator.

  In this way, Windows can properly create symbolic links when it extracts the archive. Otherwise, symbolic links will be written as normal files that contain the paths of the symbolic links as text or are empty. For more information, see the blog entry Symlinks in Windows 10!.

  If you use Git under Windows, you must enable Developer Mode or you must:
  - Set `core.symlinks` to true with the following command:

    ```
    git config --global core.symlinks true
    ```
  - Use a console that is elevated as administrator whenever you use a git command that writes to the system (for example, **git pull**, **git clone**, and **git submodule update --init --recursive**).

Overview

This tutorial contains instructions for the following getting started steps:

1. Connecting your board to a host machine.

2. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross compiling a FreeRTOS demo application to a binary image.
4. Loading the application binary image to your board, and then running the application.
5. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Set up the Microchip Curiosity PIC32MZ EF hardware

1. Connect the MikroElectronika USB UART click Board to the microBUS 1 connector on the Microchip Curiosity PIC32MZ EF.
2. Connect the PIC32 LAN8720 PHY daughter board to the J18 header on the Microchip Curiosity PIC32MZ EF.
3. Connect the MikroElectronika USB UART click Board to your computer using a USB A to USB mini-B cable.
4. To connect your board to the internet, use one of the following options:

   - To use **Wi-Fi**, connect the MikroElectronika Wi-Fi 7 click Board to the microBUS 2 connector on the Microchip Curiosity PIC32MZ EF. See Configuring the FreeRTOS demos.
   - To use **Ethernet** to connect the Microchip Curiosity PIC32MZ EF Board to the internet, connect the PIC32 LAN8720 PHY daughter board to the J18 header on the Microchip Curiosity PIC32MZ EF. Connect one end of an Ethernet cable to the LAN8720 PHY daughter board. Connect the other end to your router or other internet port. You must also define the preprocessor macro `PIC32_USE_ETHERNET`.
5. If not done already, solder the angle connector to the ICSP header on the Microchip Curiosity PIC32MZ EF.
6. Connect one end of the ICSP cable from the PICkit 3 Programming Cable Kit to the Microchip Curiosity PIC32MZ EF.

   If you don't have the PICkit 3 Programming Cable Kit, you can use M-F Dupont wire jumpers to wire the connection instead. Note that the white circle signifies the position of Pin 1.
7. Connect the other end of the ICSP cable (or jumpers) to the MPLAB Snap Debugger. Pin 1 of the 8-pin SIL Programming Connector is marked by the black triangle on the bottom right of the board.

Make sure that any cabling to Pin 1 on the Microchip Curiosity PIC32MZ EF, signified by the white circle, aligns with Pin 1 on the MPLAB Snap Debugger.

For more information about the MPLAB Snap Debugger, see the MPLAB Snap In-Circuit Debugger Information Sheet.

Set up the Microchip Curiosity PIC32MZ EF hardware using PICkit On Board (PKOB)

We recommend that you follow the setup procedure in the previous section. However, you can evaluate and run FreeRTOS demos with basic debugging using the integrated PICkit On Board (PKOB) programmer/debugger by following these steps.

1. Connect the MikroElectronika USB UART click Board to the microBUS 1 connector on the Microchip Curiosity PIC32MZ EF.
2. To connect your board to the internet, do one of the following:
   - To use **Wi-Fi**, connect the MikroElectronika Wi-Fi 7 click Board to the microBUS 2 connector on the Microchip Curiosity PIC32MZ EF. (Follow the steps "To configure your Wi-Fi" in Configuring the FreeRTOS demos.
   - To use **Ethernet** to connect the Microchip Curiosity PIC32MZ EF Board to the internet, connect the PIC32 LAN8720 PHY daughter board to the J18 header on the Microchip Curiosity PIC32MZ EF. Connect one end of an Ethernet cable to the LAN8720 PHY daughter board. Connect the other end to your router or other internet port. You must also define the preprocessor macro `PIC32_USE_ETHERNET`.
3. Connect the USB micro-B port named "USB DEBUG" on the Microchip Curiosity PIC32MZ EF Board to your computer using a USB type A to USB micro-B cable.
4. Connect the MikroElectronika USB UART click Board to your computer using a USB A to USB mini-B cable.

Set up your development environment

Note

The FreeRTOS project for this device is based on MPLAB Harmony v2. To build the project, you need to use versions of the MPLAB tools that are compatible with Harmony v2, like

v2.10 of the MPLAB XC32 Compiler and versions 2.X.X of the MPLAB Harmony Configurator (MHC).

1. Install Python version 3.x or later.
2. Install the MPLAB X IDE:

   Note

   FreeRTOS AWS Reference Integrations v202007.00 is currently supported on MPLabv5.35 only. Prior versions of FreeRTOS AWS Reference Integrations are supported on MPLabv5.40.

   MPLabv5.35 downloads

   - MPLAB X Integrated Development Environment for Windows
   - MPLAB X Integrated Development Environment for macOS
   - MPLAB X Integrated Development Environment for Linux

   Latest MPLab downloads (MPLabv5.40)

   - MPLAB X Integrated Development Environment for Windows
   - MPLAB X Integrated Development Environment for macOS
   - MPLAB X Integrated Development Environment for Linux
3. Install the MPLAB XC32 Compiler:

   - MPLAB XC32/32++ Compiler for Windows
   - MPLAB XC32/32++ Compiler for macOS
   - MPLAB XC32/32++ Compiler for Linux
4. Start up a UART terminal emulator and open a connection with the following settings:

   - Baud rate: 115200
   - Data: 8 bit
   - Parity: None

- Stop bits: 1
- Flow control: None

Monitoring MQTT messages in the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run the FreeRTOS demo project

Open the FreeRTOS demo in the MPLAB IDE

1. Open MPLAB IDE. If you have more than one version of the compiler installed, you need to select the compiler that you want to use from within the IDE.
2. From the **File** menu, choose **Open Project**.
3. Browse to and open `projects/microchip/curiosity_pic32mzef/mplab/aws_demos`.
4. Choose **Open project**.

> **Note**
>
> When you open the project for the first time, you might get an error message about the compiler. In the IDE, navigate to **Tools**, **Options**, **Embedded**, and then select the compiler that you are using for your project.

To use Ethernet to connect, you must define the preprocessor macro `PIC32_USE_ETHERNET`.

To use Ethernet to connect using the MPLAB IDE

1. In the MPLAB IDE, right-click the project and choose **Properties**.
2. In the **Project Properties** dialog box, choose ***compiler-name*** **(Global Options)** to expand it, and then select ***compiler-name*****-gcc**.
3. For **Options categories**, choose **Preprocessing and messages**, and then add the `PIC32_USE_ETHERNET` string to **Preprocessor macros**.

Run the FreeRTOS demo project

1. Rebuild your project.
2. On the **Projects** tab, right-click the `aws_demos` top-level folder, and then choose **Debug**.
3. When the debugger stops at the breakpoint in `main()`, from the **Run** menu, choose **Resume**.

Build the FreeRTOS demo with CMake

If you prefer not to use an IDE for FreeRTOS development, you can alternatively use CMake to build and run the demo applications or applications that you have developed using third-party code editors and debugging tools.

To build the FreeRTOS demo with CMake

1. Create a directory to contain the generated build files, such as *build-directory*.
2. Use the following command to generate build files from source code.

```
cmake -DVENDOR=microchip -DBOARD=curiosity_pic32mzef -DCOMPILER=xc32 -
DMCHP_HEXMATE_PATH=path/microchip/mplabx/version/mplab_platform/bin  -
DAFR_TOOLCHAIN_PATH=path/microchip/xc32/version/bin -S freertos -B build-folder
```

> **Note**
>
> You must specify the correct paths to the Hexmate and toolchain binaries, such as the `C:\Program Files (x86)\Microchip\MPLABX\v5.35\mplab_platform\bin` and `C:\Program Files\Microchip\xc32\v2.40\bin` paths.

3. Change directories to the build directory (*build-directory*), and then run `make` from that directory.

For more information, see Using CMake with FreeRTOS.

To use Ethernet to connect, you must define the preprocessor macro `PIC32_USE_ETHERNET`.

Troubleshooting

For troubleshooting information, see Troubleshooting getting started.

Getting started with the Nordic nRF52840-DK

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Nordic nRF52840-DK. If you do not have the Nordic nRF52840-DK, visit the AWS Partner Device Catalog to purchase one from our partner.

Before you begin, you need to Set up AWS IoT and Amazon Cognito for FreeRTOS Bluetooth Low Energy.

To run the FreeRTOS Bluetooth Low Energy demo, you also need an iOS or Android mobile device with Bluetooth and Wi-Fi capabilities.

> **Note**
>
> If you are using an iOS device, you need Xcode to build the demo mobile application. If you are using an Android device, you can use Android Studio to build the demo mobile application.

Overview

This tutorial contains instructions for the following getting started steps:

1. Connecting your board to a host machine.

2. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross compiling a FreeRTOS demo application to a binary image.
4. Loading the application binary image to your board, and then running the application.
5. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Set up the Nordic hardware

Connect your host computer to the USB port labeled J2, located directly above the coin cell battery holder on your Nordic nRF52840 board.

For more information about setting up the Nordic nRF52840-DK, see the nRF52840 Development Kit User Guide.

Set up your development environment

Download and install Segger Embedded Studio

FreeRTOS supports Segger Embedded Studio as a development environment for the Nordic nRF52840-DK.

To set up your environment, you need to download and install Segger Embedded Studio on your host computer.

To download and install Segger Embedded Studio

1. Go to the Segger Embedded Studio Downloads page, and choose the Embedded Studio for ARM option for your operating system.
2. Run the installer and follow the prompts to completion.

Set up the FreeRTOS Bluetooth Low Energy Mobile SDK demo application

To run the FreeRTOS demo project across Bluetooth Low Energy, you need to run the FreeRTOS Bluetooth Low Energy Mobile SDK demo application on your mobile device.

To set up the FreeRTOS Bluetooth Low Energy Mobile SDK Demo application

1. Follow the instructions in Mobile SDKs for FreeRTOS Bluetooth devices to download and install the SDK for your mobile platform on your host computer.

2. Follow the instructions in FreeRTOS Bluetooth Low Energy Mobile SDK demo application to set up the demo mobile application on your mobile device.

Establish a serial connection

Segger Embedded Studio includes a terminal emulator that you can use to receive log messages across a serial connection to your board.

To establish a serial connection with Segger Embedded Studio

1. Open Segger Embedded Studio.
2. From the top menu, choose **Target**, **Connect J-Link**.
3. From the top menu, choose **Tools**, **Terminal Emulator**, **Properties**, and set the properties as instructed in Installing a terminal emulator.
4. From the top menu, choose **Tools**, **Terminal Emulator**, **Connect *port* (115200,N,8,1)**.

> **Note**
>
> The Segger embedded studio terminal emulator does not support an input capability. For this, use a terminal emulator like PuTTy, Tera Term, or GNU Screen. Configure the terminal to connect to your board by a serial connection as instructed in Installing a terminal emulator.

Download and configure FreeRTOS

After you set up your hardware and environment, you can download FreeRTOS.

Download FreeRTOS

To download FreeRTOS for the Nordic nRF52840-DK, go to the FreeRTOS GitHub page and clone the repository. See the README.md file for instructions.

> **Important**
>
> - In this topic, the path to the FreeRTOS download directory is referred to as *freertos*.
> - Space characters in the *freertos* path can cause build failures. When cloning or copying the repository, make sure the path you create does not contain space characters.

- The maximum length of a file path on Microsoft Windows is 260 characters. Long FreeRTOS download directory paths can cause build failures.
- Because the source code may contain symbolic links, if you're using Windows to extract the archive, you may have to:
  - Enable Developer Mode or,
  - Use a console that is elevated as administrator.

  In this way, Windows can properly create symbolic links when it extracts the archive. Otherwise, symbolic links will be written as normal files that contain the paths of the symbolic links as text or are empty. For more information, see the blog entry Symlinks in Windows 10!.

  If you use Git under Windows, you must enable Developer Mode or you must:
  - Set `core.symlinks` to true with the following command:

    ```
    git config --global core.symlinks true
    ```
  - Use a console that is elevated as administrator whenever you use a git command that writes to the system (for example, **git pull**, **git clone**, and **git submodule update --init --recursive**).

Configure your project

To enable the demo, you need to configure your project to work with AWS IoT. To configure your project to work with AWS IoT, your device must be registered as an AWS IoT thing. You should have registered your device when you Set up AWS IoT and Amazon Cognito for FreeRTOS Bluetooth Low Energy.

To configure your AWS IoT endpoint

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Settings**.

   Your AWS IoT endpoint appears in the **Device data endpoint** text box. 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**. Make a note of the AWS IoT thing name for your device.

4. With your AWS IoT endpoint and your AWS IoT thing name on hand, open *freertos*/demos/include/aws_clientcredential.h in your IDE, and specify values for the following #define constants:

   - clientcredentialMQTT_BROKER_ENDPOINT *Your AWS IoT endpoint*
   - clientcredentialIOT_THING_NAME *Your board's AWS IoT thing name*

To enable the demo

1. Check that the Bluetooth Low Energy GATT Demo is enabled. Go to vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/iot_ble_config.h, and add #define IOT_BLE_ADD_CUSTOM_SERVICES (1) to the list of define statements.
2. Open vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/aws_demo_config.h, and define either CONFIG_OTA_MQTT_BLE_TRANSPORT_DEMO_ENABLED or CONFIG_OTA_HTTP_BLE_TRANSPORT_DEMO_ENABLED as in this example.

```
/* To run a particular demo you need to define one of these.
 * Only one demo can be configured at a time
 *
 *          CONFIG_BLE_GATT_SERVER_DEMO_ENABLED
 *          CONFIG_MQTT_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_SHADOW_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_OTA_MQTT_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_OTA_HTTP_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_POSIX_DEMO_ENABLED
 *
 *  These defines are used in iot_demo_runner.h for demo selection */

#define CONFIG_OTA_MQTT_BLE_TRANSPORT_DEMO_ENABLED
```

3. Since the Nordic chip comes with very little RAM (250KB), the BLE configuration might need to be changed to allow for larger GATT table entries compared to the size of each attribute. In this way you can adjust the amount of memory the application gets. To do this, override the definitions of the following attributes in the file *freertos*/vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/sdk_config.h:

   - NRF_SDH_BLE_VS_UUID_COUNT

The number of vendor-specific UUIDs. Increase this count by 1 when you add a new vendor-specific UUID.

- NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE

  Attribute Table size in bytes. The size must be a multiple of 4. This value indicates the set amount of memory dedicated for the attribute table (including the characteristic size), so this will vary from project to project. If you exceed the size of the attribute table you will get a NRF_ERROR_NO_MEM error. If you modify the NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE usually you must also reconfigure the RAM settings.

(For tests, the location of the file is *freertos*`/vendors/nordic/boards/nrf52840-dk/aws_tests/config_files/sdk_config.h`.)

Build and run the FreeRTOS demo project

After you download FreeRTOS and configure your demo project, you are ready to build and run the demo project on your board.

> **Important**
>
> If this is the first time that you are running the demo on this board, you need to flash a bootloader to the board before the demo can run.
> To build and flash the bootloader, follow the steps below, but instead of using the `projects/nordic/nrf52840-dk/ses/aws_demos/aws_demos.emProject` project file, use `projects/nordic/nrf52840-dk/ses/aws_demos/bootloader/bootloader.emProject`.

To build and run the FreeRTOS Bluetooth Low Energy demo from Segger Embedded Studio

1. Open Segger Embedded Studio. From the top menu, choose **File**, choose **Open Solution**, and then navigate to the project file `projects/nordic/nrf52840-dk/ses/aws_demos/aws_demos.emProject`
2. If you are using the Segger Embedded Studio terminal emulator, choose **Tools** from the top menu, and then choose **Terminal Emulator**, **Terminal Emulator** to display information from your serial connection.

If you are using another terminal tool, you can monitor that tool for output from your serial connection.

3. Right-click the `aws_demos` demo project in the **Project Explorer**, and choose **Build**.

> **Note**
>
> If this is your first time using Segger Embedded Studio, you might see you a warning "No license for commercial use". Segger Embedded Studio can be used free of charge for Nordic Semiconductor devices. Request a free license then, during setup choose **Activate Your Free License**, and follow the instructions.

4. Choose **Debug**, and then choose **Go**.

   After the demo starts, it waits to pair with a mobile device across Bluetooth Low Energy.

5. Follow the instructions for the MQTT over Bluetooth Low Energy Demo Application to complete the demo with the FreeRTOS Bluetooth Low Energy Mobile SDK demo application as the mobile MQTT proxy.

Troubleshooting

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Nuvoton NuMaker-IoT-M487

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Nuvoton NuMaker-IoT-M487 development board. The series microcontroller, and includes built-in RJ45 Ethernet and Wi-Fi modules. If you don't have the Nuvoton NuMaker-IoT-M487, visit the AWS Partner Device Catalog to purchase one from our partner.

Before you begin, you must configure AWS IoT and your FreeRTOS software to connect your development board to the AWS Cloud. For instructions, see First steps. In this tutorial, the path to the FreeRTOS download directory is referred to as *freertos*.

Overview

This tutorial guides you through the following steps:

1. Install software on your host machine for developing and debugging embedded applications for your microcontroller board.
2. Cross-compile a FreeRTOS demo application to a binary image.
3. Load the application binary image to your board, and then run the application.

Set up your development environment

The Keil MDK Nuvoton edition is designed for developing and debugging applications for Nuvoton M487 boards. The Keil MDK v5 Essential, Plus, or Pro version should also work for the Nuvoton M487 (Cortex-M4 core) MCU. You can download the Keil MDK Nuvoton edition with a price discount for the Nuvoton Cortex-M4 series MCUs. The Keil MDK is only supported on Windows.

To install the development tool for the NuMaker-IoT-M487

1. Download the Keil MDK Nuvoton Edition from the Keil MDK website.
2. Install the Keil MDK on your host machine using your license. The Keil MDK includes the Keil µVision IDE, a C/C++ compilation toolchain, and the µVision debugger.

   If you experience issues during installation, contact Nuvoton for assistance.
3. Install the **Nu-Link_Keil_Driver_V3.06.7215r** (or latest version), which is on the Nuvoton Development Tool page.

Build and run the FreeRTOS demo project

To build the FreeRTOS demo project

1. Open the Keil µVision IDE.
2. On the **File** menu, choose **Open**. In the **Open file** dialog box, make sure the file type selector is set to **Project Files**.
3. Choose either the Wi-Fi or Ethernet demo project to build.

- To open the Wi-Fi demo project, choose the target project `aws_demos.uvproj` in the *`freertos`*`\projects\nuvoton\numaker_iot_m487_wifi\uvision\aws_demos` directory.
- To open the Ethernet demo project, choose the target project `aws_demos_eth.uvproj` in the *`freertos`*`\projects\nuvoton\numaker_iot_m487_wifi\uvision\aws_demos_eth` directory.

4. To make sure your settings are correct to flash the board, right-click the `aws_demo` project in the IDE, and then choose **Options**. (See Troubleshooting for more details.)
5. On the **Utilities** tab, verify that **Use Target Driver for Flash Programming** is selected, and that **Nuvoton Nu-Link Debugger** is set as the target driver.
6. On the **Debug** tab, next to **Nuvoton Nu-Link Debugger**, choose **Settings**.
7. Verify that the **Chip Type** is set to **M480**.
8. In the Keil µVision IDE **Project** navigation pane, choose the `aws_demos` project. On the **Project** menu, choose **Build Target**.

You can use the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***`your-thing-name`*`/example/topic`**, and then choose **Subscribe to topic**.

To run the FreeRTOS demo project

1. Connect your Numaker-IoT-M487 board to your host machine (computer).
2. Rebuild the project.
3. In the Keil µVision IDE, on the **Flash** menu, choose **Download**.
4. On the **Debug** menu, choose **Start/Stop Debug Session**.
5. When the debugger stops at the breakpoint in `main()`, open the **Run** menu, and then choose **Run (F5)**.

You should see MQTT messages sent by your device in the MQTT client in the AWS IoT console.

Using CMake with FreeRTOS

You can also use CMake to build and run the FreeRTOS demo applications or applications you have developed using third-party code editors and debugging tools.

Make sure you have installed the CMake build system. Follow the instructions in Using CMake with FreeRTOS, and then follow the steps in this section.

> **Note**
>
> Be sure the path to the location of the compiler (Keil) is in your Path system variable, for example, `C:\Keil_v5\ARM\ARMCC\bin`.

You can also use the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

To generate build files from source files and run the demo project

1. On your host machine, open the command prompt and navigate to the *`freertos`* folder.
2. Create a folder to contain the generated build file. We will refer to this folder as the *`BUILD_FOLDER`*.
3. Generate the build files for either the Wi-Fi or Ethernet demo.

   - For Wi-Fi:

     Navigate to the directory that contains the source files for the FreeRTOS demo project. Then, generate the build files by running the following command.

```
cmake -DVENDOR=nuvoton -DBOARD=numaker_iot_m487_wifi -DCOMPILER=arm-keil -S . -
B BUILD_FOLDER -G Ninja
```

- For Ethernet:

  Navigate to the directory that contains the source files for the FreeRTOS demo project. Then, generate the build files by running the following command.

```
cmake -DVENDOR=nuvoton -DBOARD=numaker_iot_m487_wifi -DCOMPILER=arm-keil -
DAFR_ENABLE_ETH=1 -S . -B BUILD_FOLDER -G Ninja
```

4. Generate the binary to flash onto the M487 by running the following command.

```
cmake --build BUILD_FOLDER
```

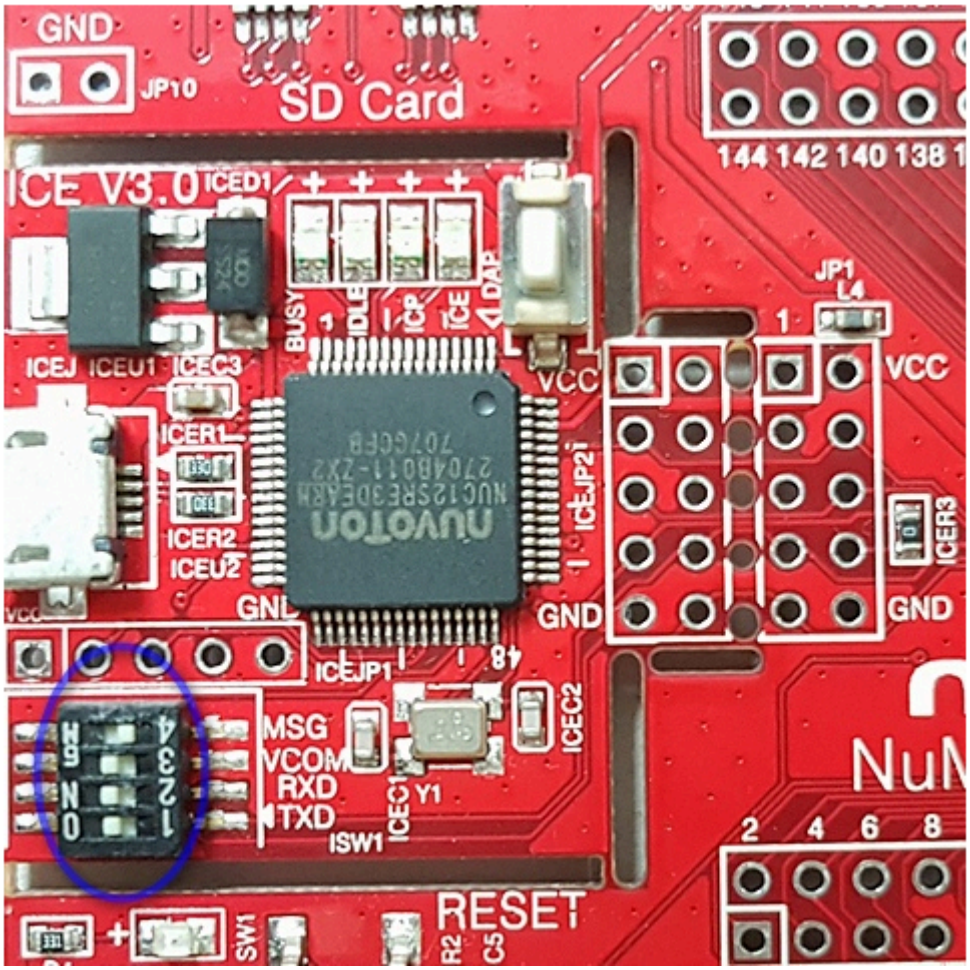
   At this point, the binary file `aws_demos.bin` should be in the *BUILD_FOLDER*`/vendors/Nuvoton/boards/numaker_iot_m487_wifi` folder.

5. To configure the board for flashing mode, make sure the MSG switch (No.4 of ISW1 on ICE) is switched ON. When you plug in the board, a window (and drive) will be assigned. (See Troubleshooting.)
6. Open a terminal emulator to view the messages over UART. Follow the instructions at Installing a terminal emulator.
7. Run the demo project by copying the generated binary onto the device.

   If you subscribed to the MQTT topic with the AWS IoT MQTT client, you should see MQTT messages sent by your device in the AWS IoT console.

Troubleshooting

- If your windows can't recognize the device `VCOM`, install the NuMaker windows serial port driver from the link Nu-Link USB Driver v1.6.
- If you connect your device to the Keil MDK (IDE) through Nu-Link, make sure the MSG switch (No.4 of ISW1 on ICE) is OFF, as shown.



If you experience issues setting up your development environment or connecting to your board, contact Nuvoton.

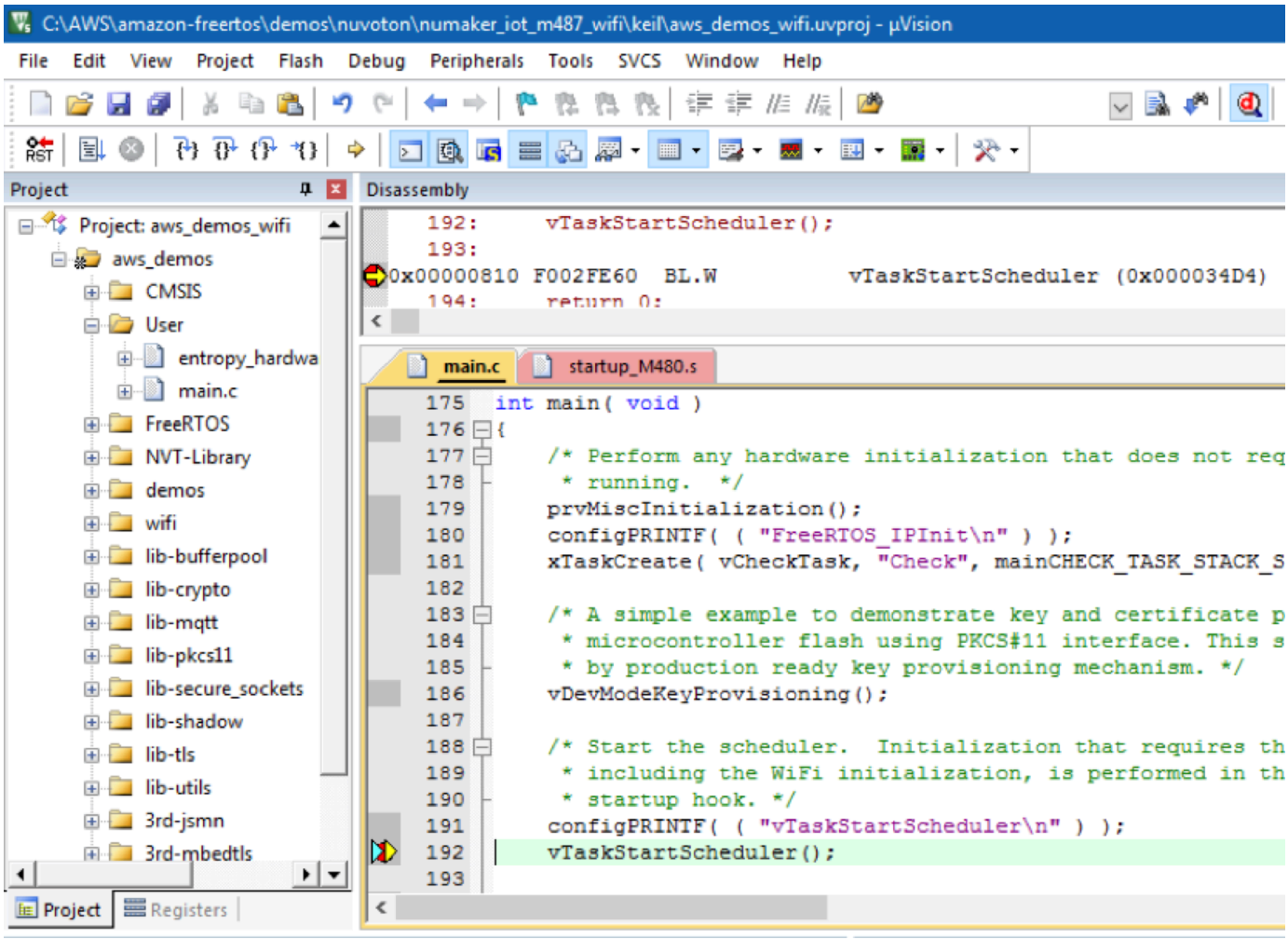
Debugging FreeRTOS projects in Keil µVision

To start a debug session in Keil µVision

1. Open Keil µVision.
2. Follow the steps to build the FreeRTOS demo project in Build and run the FreeRTOS demo project.
3. On the **Debug** menu, choose **Start/Stop Debug Session**.

   The **Call Stack + Locals** window appears when you start a debug session. µVision flashes the demo to the board, runs the demo, and stops at the beginning of the `main()` function.

4. Set breakpoints in your project's source code, and then run the code. The project should look something like the following.

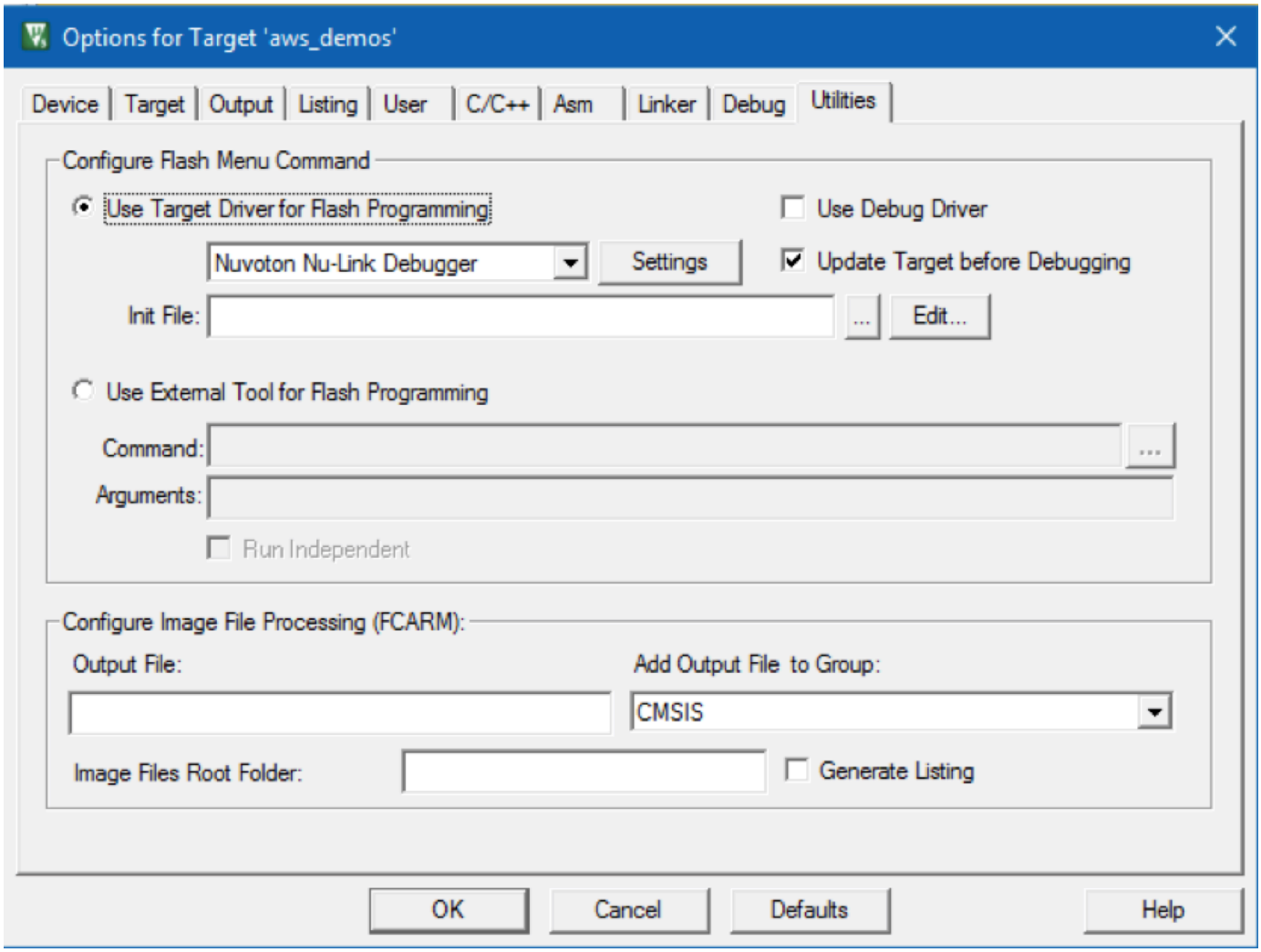


Troubleshooting µVision debug settings

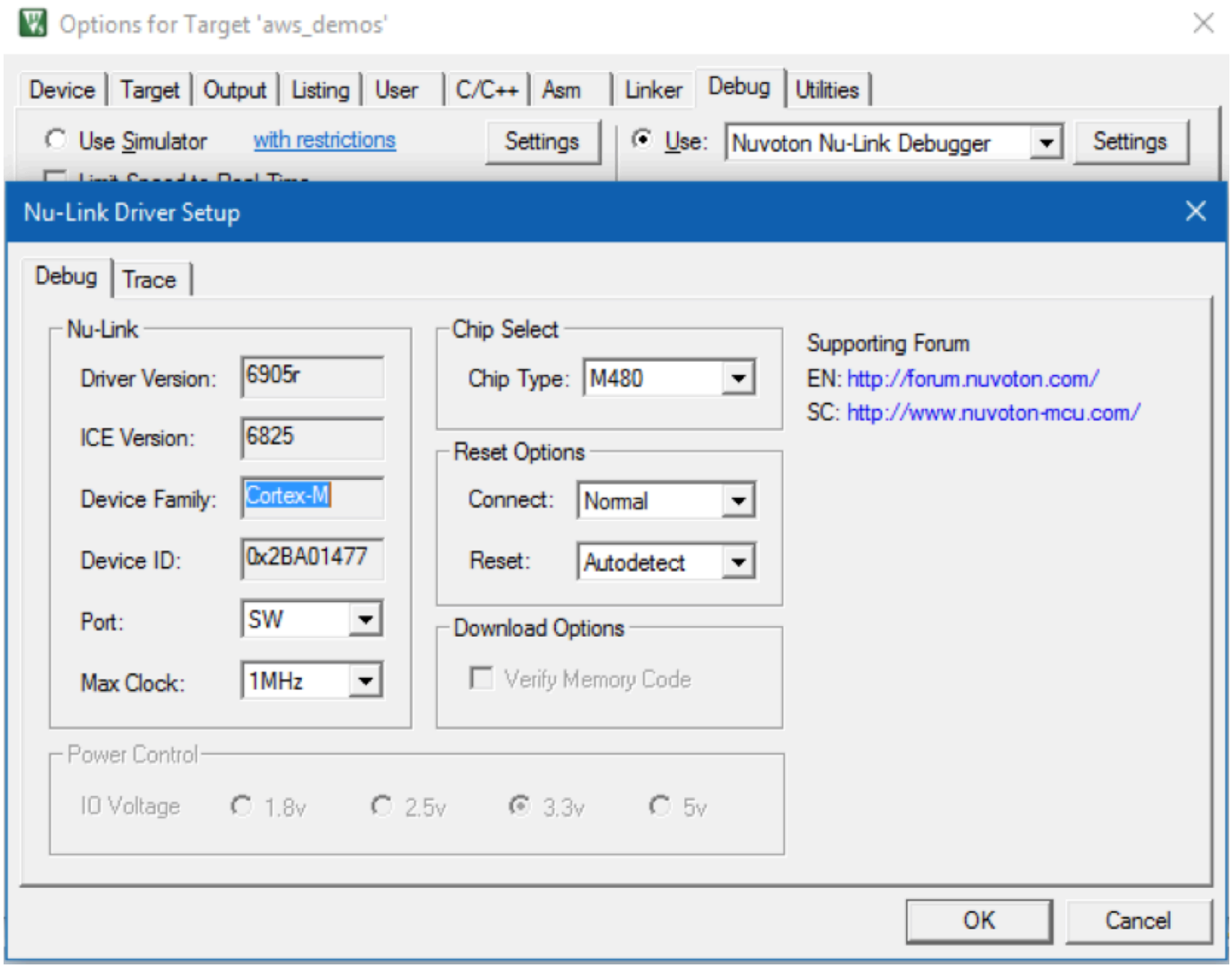
If you encounter problems while debugging an application, check that your debug settings are set correctly in Keil µVision.

To verify that the µVision debug settings are correct

1. Open Keil µVision.
2. Right-click the aws_demo project in the IDE, and then choose **Options**.
3. On the **Utilities** tab, verify that **Use Target Driver for Flash Programming** is selected, and that **Nuvoton Nu-Link Debugger** is set as the target driver.



4. On the **Debug** tab, next to **Nuvoton Nu-Link Debugger**, choose **Settings**.



5. Verify that the **Chip Type** is set to **M480**.

Getting started with the NXP LPC54018 IoT Module

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the NXP LPC54018 IoT Module. If you do not have an NXP LPC54018 IoT Module, visit the AWS Partner Device Catalog to purchase one from our partner. Use a USB cable to connect your NXP LPC54018 IoT Module to your computer.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains instructions for the following getting started steps:

1. Connecting your board to a host machine.
2. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross compiling a FreeRTOS demo application to a binary image.
4. Loading the application binary image to your board, and then running the application.

Set up the NXP hardware

To set up the NXP LPC54018

- Connect your computer to the USB port on the NXP LPC54018.

To set up the JTAG debugger

You need a JTAG debugger to launch and debug your code running on the NXP LPC54018 board. FreeRTOS was tested using an OM40006 IoT Module. For more information about supported debuggers, see the *User Manual for NXP LPC54018 IoT Module* that is available from the OM40007 LPC54018 IoT Module product page.

1. If you're using an OM40006 IoT Module debugger, use a converter cable to connect the 20-pin connector from the debugger to the 10-pin connector on the NXP IoT module.
2. Connect the NXP LPC54018 and the OM40006 IoT Module Debugger to the USB ports on your computer using mini-USB to USB cables.

Set up your development environment

FreeRTOS supports two IDEs for the NXP LPC54018 IoT Module: IAR Embedded Workbench and MCUXpresso.

Before you begin, install one of these IDEs.

To install IAR Embedded Workbench for ARM

1. Browse to IAR Embedded Workbench for ARM and download the software.

   > **Note**
   >
   > IAR Embedded Workbench for ARM requires Microsoft Windows.

2. Run the installer and follow the prompts.
3. In the **License Wizard**, choose **Register with IAR Systems to get an evaluation license**.
4. Put the bootloader on the device before attempting to run any demos.

To install MCUXpresso from NXP

1. Download and run the MCUXpresso installer from NXP.

   > **Note**
   >
   > Versions 10.3.x and later are supported.

2. Browse to MCUXpresso SDK and choose **Build your SDK.**

   > **Note**
   >
   > Versions 2.5 and later are supported.

3. Choose **Select Development Board**.
4. Under **Select Development Board**, in **Search by Name**, enter **`LPC54018-IoT-Module`**.
5. Under **Boards**, choose **LPC54018-IoT-Module**.
6. Verify the hardware details, and then choose **Build MCUXepresso SDK**.

7. The SDK for Windows using the MCUXpresso IDE is already built. Choose **Download SDK**. If you are using another operating system, under **Host OS**, choose your operating system, and then choose **Download SDK**.
8. Start the MCUXpresso IDE, and choose the **Installed SDKs** tab.
9. Drag and drop the downloaded SDK archive file into the **Installed SDKs** window.

If you experience issues during installation, see NXP Support or NXP Developer Resources.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*****/example/topic**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run the FreeRTOS Demo project

Import the FreeRTOS demo into your IDE

To import the FreeRTOS sample code into the IAR Embedded Workbench IDE

1. Open IAR Embedded Workbench, and from the **File** menu, choose **Open Workspace**.
2. In the **search-directory** text box, enter `projects/nxp/lpc54018iotmodule/iar/aws_demos`, and choose **aws_demos.eww**.
3. From the **Project** menu, choose **Rebuild All**.

To import the FreeRTOS sample code into the MCUXpresso IDE

1. Open MCUXpresso, and from the **File** menu, choose **Open Projects From File System**.

2. In the **Directory** text box, enter `projects/nxp/lpc54018iotmodule/mcuxpresso/aws_demos`, and choose **Finish**
3. From the **Project** menu, choose **Build All**.

Run the FreeRTOS demo project

To run the FreeRTOS demo project with the IAR Embedded Workbench IDE

1. In your IDE, from the **Project** menu, choose **Make**.
2. From the **Project** menu, choose **Download and Debug**.
3. From the **Debug** menu, choose **Start Debugging**.
4. When the debugger stops at the breakpoint in `main`, from the **Debug** menu, choose **Go.**

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

To run the FreeRTOS demo project with the MCUxpresso IDE

1. In your IDE, from the **Project** menu, choose **Build**.
2. If this is your first time debugging, choose the `aws_demos` project and from the **Debug** toolbar, choose the blue debug button.
3. Any detected debug probes are displayed. Choose the probe you want to use, and then choose **OK** to start debugging.

> **Note**
>
> When the debugger stops at the breakpoint in `main()`, press the debug restart button
>
> 
>
> once to reset the debugging session. (This is required due to a bug with MCUXpresso debugger for NXP54018-IoT-Module).

4. When the debugger stops at the breakpoint in `main()`, from the **Debug** menu, choose **Go**.

Troubleshooting

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Renesas Starter Kit+ for RX65N-2MB

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Renesas Starter Kit+ for RX65N-2MB. If you do not have the Renesas RSK+ for RX65N-2MB, visit the AWS Partner Device Catalog, and purchase one from our partners.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains instructions for the following getting started steps:

1. Connecting your board to a host machine.
2. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross compiling a FreeRTOS demo application to a binary image.
4. Loading the application binary image to your board, and then running the application.

Set up the Renesas hardware

To set up the RSK+ for RX65N-2MB

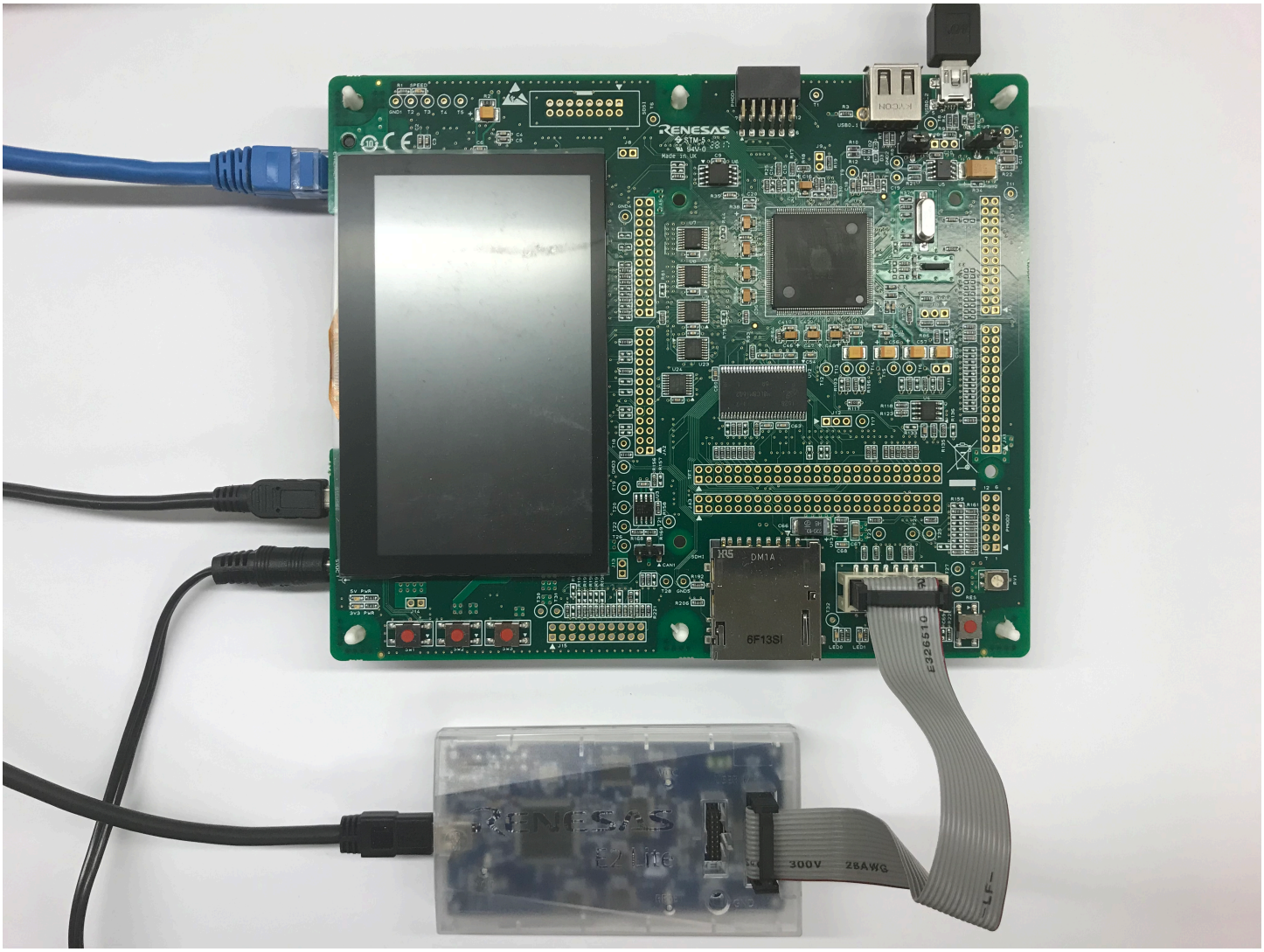
1. Connect the positive +5V power adapter to the PWR connector on the RSK+ for RX65N-2MB.

2. Connect your computer to the USB2.0 FS port on the RSK+ for RX65N-2MB.
3. Connect your computer to the USB-to-serial port on the RSK+ for RX65N-2MB.
4. Connect a router or internet-connected Ethernet port to the Ethernet port on the RSK+ for RX65N-2MB.

To set up the E2 Lite Debugger module

1. Use the 14-pin ribbon cable to connect the E2 Lite Debugger module to the ‘E1/E2 Lite’ port on the RSK+ for RX65N-2MB.
2. Use a USB cable to connect the E2 Lite debugger module to your host machine. When the E2 Lite debugger is connected to both the board and your computer, a green ‘ACT’ LED on the debugger flashes.
3. After the debugger is connected to your host machine and RSK+ for RX65N-2MB, the E2 Lite debugger drivers begin installing.

   Note that administrator privileges are required to install the drivers.



Set up your development environment

To set up FreeRTOS configurations for the RSK+ for RX65N-2MB, use the Renesas e^{2}studio IDE and CC-RX compiler.

> **Note**
>
> The Renesas e^{2}studio IDE and CC-RX compiler are only supported on Windows 7, 8, and 10 operating systems.

To download and install e^{2}studio

1. Go to the Renesas e^{2}studio installer download page, and download the offline installer.

2. You are directed to a Renesas Login page.

   If you have an account with Renesas, enter your sign-in credentials and then choose **Login**.

   If you do not have an account, choose **Register now**, and follow the first registration steps. You should receive an email with a link to activate your Renesas account. Follow this link to complete your registration with Renesas, and then log in to Renesas.
3. After you log in, download the e^{2}studio installer to your computer.
4. Open the installer and follow the steps to completion.

For more information, see the e^{2}studio on the Renesas website.

To download and install the RX Family C/C++ Compiler Package

1. Go to the RX Family C/C++ Compiler Package download page, and download the V3.00.00 package.
2. Open the executable and install the compiler.

For more information, see the C/C++ Compiler Package for RX Family on the Renesas website.

> **Note**
>
> The compiler is available free for evaluation version only and valid for 60 days. On the 61st day, you need to get a License Key. For more information, see Evaluation Software Tools.

Build and run FreeRTOS samples

Now that you have configured the demo project, you are ready to build and run the project on your board.

Build the FreeRTOS Demo in e^{2}studio

To import and build the demo in e^{2}studio

1. Launch e^{2}studio from the Start menu.
2. On the **Select a directory as a workspace** window, browse to the folder that you want to work in, and choose **Launch**.

3. The first time you open e^{2}studio, the **Toolchain Registry** window opens. Choose **Renesas Toolchains**, and confirm that **`CC-RX v3.00.00`** is selected. Choose **Register**, and then choose **OK**.
4. If you are opening e^{2}studio for the first time, the **Code Generator Registration** window appears. Choose **OK**.
5. The **Code Generator COM component register** window appears. Under **Please restart e^{2}studio to use Code Generator**, choose **OK**.
6. The **Restart e^{2}studio** window appears. Choose **OK**.
7. e^{2}studio restarts. On the **Select a directory as a workspace** window, choose **Launch**.
8. On the e^{2}studio welcome screen, choose the **Go to the e^{2}studio workbench** arrow icon.
9. Right-click the **Project Explorer** window, and choose **Import**.
10. In the import wizard, choose **General**, **Existing Projects into Workspace**, and then choose **Next**.
11. Choose **Browse**, locate the directory `projects/renesas/rx65n-rsk/e2studio/aws_demos`, and then choose **Finish**.
12. From **Project** menu, choose **Project**, **Build All**.

    The build console issues a warning message that the License Manager is not installed. You can ignore this message unless you have a license key for the CC-RX compiler. To install the License Manager, see the License Manager download page.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***`your-thing-name`*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Run the FreeRTOS project

To run the project in e^{2}studio

1. Confirm that you have connected the E2 Lite Debugger module to your RSK+ for RX65N-2MB
2. From the top menu, choose **Run**, **Debug Configuration**.
3. Expand **Renesas GDB Hardware Debugging**, and choose **aws_demos HardwareDebug**.
4. Choose the **Debugger** tab, and then choose the **Connection Settings** tab. Confirm that your connection settings are correct.
5. Choose **Debug** to download the code to your board and begin debugging.

   You might be prompted by a firewall warning for `e2-server-gdb.exe`. Check **Private networks, such as my home or work network**, and then choose **Allow access**.
6. e^{2}studio might ask to change to **Renesas Debug Perspective**. Choose **Yes**.

   The green 'ACT' LED on the E2 Lite Debugger illuminates.
7. After the code is downloaded to the board, choose **Resume** to run the code up to the first line of the main function. Choose **Resume** again to run the rest of the code.

For the latest projects released by Renesas, see the `renesas-rx` fork of the `amazon-freertos` repository on GitHub.

Troubleshooting

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the STMicroelectronics STM32L4 Discovery Kit IoT Node

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the STMicroelectronics STM32L4 Discovery Kit IoT Node. If you do not already have the STMicroelectronics STM32L4 Discovery Kit IoT Node, visit the AWS Partner Device Catalog to purchase one from our partner.

Make sure you have installed the latest Wi-Fi firmware. To download the latest Wi-Fi firmware, see STM32L4 Discovery kit IoT node, low-power wireless, Bluetooth Low Energy, NFC, SubGHz, Wi-Fi. Under **Binary Resources**, choose **Inventek ISM 43362 Wi-Fi module firmware update (read the readme file for instructions)** .

Before you begin, you must configure AWS IoT, your FreeRTOS download, and Wi-Fi to connect your device to the AWS Cloud. See First steps for instructions . In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains instructions for the following getting started steps:

1. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
2. Cross compiling a FreeRTOS demo application to a binary image.
3. Loading the application binary image to your board, and then running the application.

Set up your development environment

Install System Workbench for STM32

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

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

Build and run the FreeRTOS demo project

Import the FreeRTOS demo into the STM32 System Workbench

1. Open the STM32 System Workbench and enter a name for a new workspace.

2. From the **File** menu, choose **Import**. Expand **General**, choose **Existing Projects into Workspace**, and then choose **Next**.
3. In **Select Root Directory**, enter `projects/st/stm32l475_discovery/ac6/aws_demos`.
4. The project `aws_demos` should be selected by default.
5. Choose **Finish** to import the project into STM32 System Workbench.
6. From the **Project** menu, choose **Build All**. Confirm the project compiles without any errors.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Run the FreeRTOS demo project

1. Use a USB cable to connect your STMicroelectronics STM32L4 Discovery Kit IoT Node to your computer. (Check the manufacturer's documentation that came with your board for the correct USB port to use.)
2. From **Project Explorer**, right-click `aws_demos`, choose **Debug As**, and then choose **Ac6 STM32 C/C++ Application**.

   If a debug error occurs the first time a debug session is launched, follow these steps:

   1. In STM32 System Workbench, from the **Run** menu, choose **Debug Configurations**.
   2. Choose **aws_demos Debug**. (You might need to expand **Ac6 STM32 Debugging**.)
   3. Choose the **Debugger** tab.
   4. In **Configuration Script**, choose **Show Generator Options**.

5. In **Mode Setup**, set **Reset Mode** to **Software System Reset**. Choose **Apply**, and then choose **Debug**.

3. When the debugger stops at the breakpoint in `main()`, from the **Run** menu, choose **Resume**.

Using CMake with FreeRTOS

If you prefer not to use an IDE for FreeRTOS development, you can alternatively use CMake to build and run the demo applications or applications that you have developed using third-party code editors and debugging tools.

First create a folder to contain the generated build files (*`build-folder`*).

Use the following command to generate build files:

```
cmake -DVENDOR=st -DBOARD=stm32l475_discovery -DCOMPILER=arm-gcc -S freertos -B build-
folder
```

If `arm-none-eabi-gcc` is not in your shell path, you also need to set the `AFR_TOOLCHAIN_PATH` CMake variable. For example:

```
-D AFR_TOOLCHAIN_PATH=/home/user/opt/gcc-arm-none-eabi/bin
```

For more information about using CMake with FreeRTOS, see Using CMake with FreeRTOS.

Troubleshooting

If you see the following in the UART output from the demo application, you need to update the Wi-Fi module's firmware:

```
[Tmr Svc] WiFi firmware version is: xxxxxxxxxxxx
[Tmr Svc] [WARN] WiFi firmware needs to be updated.
```

To download the latest Wi-Fi firmware, see STM32L4 Discovery kit IoT node, low-power wireless, Bluetooth Low Energy, NFC, SubGHz, Wi-Fi. In **Binary Resources**, choose the download link for **Inventek ISM 43362 Wi-Fi module firmware update**.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Texas Instruments CC3220SF-LAUNCHXL

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Texas Instruments CC3220SF-LAUNCHXL. If you do not have the Texas Instruments (TI) CC3220SF-LAUNCHXL Development Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

Overview

This tutorial contains instructions for the following getting started steps:

1. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
2. Cross compiling a FreeRTOS demo application to a binary image.
3. Loading the application binary image to your board, and then running the application.

Set up your development environment

Follow the steps below to set up your development environment to get started with FreeRTOS.

Note that FreeRTOS supports two IDEs for the TI CC3220SF-LAUNCHXL Development Kit: Code Composer Studio and IAR Embedded Workbench version 8.32. You can use either IDE to get started.

Install Code Composer Studio

1. Browse to TI Code Composer Studio.
2. Download the offline installer for the platform of your host machine (Windows, macOS, or Linux 64-bit).

3. Unzip and run the offline installer. Follow the prompts.
4. For **Product Families to Install**, choose **SimpleLink Wi-Fi CC32xx Wireless MCUs**.
5. On the next page, accept the default settings for debugging probes, and then choose **Finish**.

If you experience issues when you are installing Code Composer Studio, see TI Development Tools Support, Code Composer Studio FAQs, and Troubleshooting CCS.

Install IAR Embedded Workbench

1. Download and run the Windows installer for version 8.32 of the IAR Embedded Workbench for ARM. In **Debug probe drivers**, make sure that **TI XDS** is selected.
2. Complete the installation and launch the program. On the **License Wizard** page, choose **Register with IAR Systems to get an evaluation license**, or use your own IAR license.

Install the SimpleLink CC3220 SDK

Install the SimpleLink CC3220 SDK. The SimpleLink Wi-Fi CC3220 SDK contains drivers for the CC3220SF programmable MCU, more than 40 sample applications, and documentation required to use the samples.

Install Uniflash

Install Uniflash. CCS Uniflash is a standalone tool used to program on-chip flash memory on TI MCUs. Uniflash has a GUI, command line, and scripting interface.

Install the latest service pack

1. On your TI CC3220SF-LAUNCHXL, place the SOP jumper on the middle set of pins (position = 1) and reset the board.
2. Start Uniflash. If your CC3220SF LaunchPad board appears under **Detected Devices**, choose **Start**. If your board is not detected, choose **CC3220SF-LAUNCHXL** from the list of boards under **New Configuration**, and then choose **Start Image Creator**.
3. Choose **New Project**.
4. On the **Start new project** page, enter a name for your project. For **Device Type**, choose **CC3220SF**. For **Device Mode**, choose **Develop**, and then choose **Create Project**.
5. On the right side of the Uniflash application window, choose **Connect**.
6. From the left column, choose **Advanced**, **Files**, and then **Service Pack**.

7. Choose **Browse**, and then navigate to where you installed the CC3220SF SimpleLink SDK. The service pack is located at `ti/simplelink_cc32xx_sdk_`*`VERSION`*`/tools/cc32xx_tools/servicepack-cc3x20/sp_`*`VERSION`*`.bin`.
8. Choose the **Burn**

   ()

   button, and then choose **Program Image (Create & Program)** to install the service pack. Remember to switch the SOP jumper back to position 0 and reset the board.

Configure Wi-Fi provisioning

To configure the Wi-Fi settings for your board, do one of the following:

- Configure the FreeRTOS demo application described in Configuring the FreeRTOS demos.
- Use SmartConfig from Texas Instruments.

Build and run the FreeRTOS demo project

Build and run the FreeRTOS demo project in TI Code Composer

To import the FreeRTOS demo into TI Code Composer

1. Open TI Code Composer, and choose **OK** to accept the default workspace name.
2. On the **Getting Started** page, choose **Import Project**.
3. In **Select search-directory**, enter `projects/ti/cc3220_launchpad/ccs/aws_demos`. The project `aws_demos` should be selected by default. To import the project into TI Code Composer, choose **Finish**.
4. In **Project Explorer**, double-click **aws_demos** to make the project active.
5. From **Project**, choose **Build Project** to make sure the project compiles without errors or warnings.

To run the FreeRTOS demo in TI Code Composer

1. Make sure the Sense On Power (SOP) jumper on your Texas Instruments CC3220SF-LAUNCHXL is in position 0. For more information, see SimpleLink Wi-Fi CC3x20, CC3x3x Network Processor User's Guide.

2. Use a USB cable to connect your Texas Instruments CC3220SF-LAUNCHXL to your computer.
3. In the project explorer, make sure the `CC3220SF.ccxml` is selected as the active target configuration. To make it active, right-click the file and choose **Set as active target configuration**.
4. In TI Code Composer, from **Run**, choose **Debug**.
5. When the debugger stops at the breakpoint in `main()`, go to the **Run** menu, and choose **Resume**.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run FreeRTOS demo project in IAR Embedded Workbench

To import the FreeRTOS demo into IAR Embedded Workbench

1. Open IAR Embedded Workbench, choose **File**, and then choose **Open Workspace**.
2. Navigate to `projects/ti/cc3220_launchpad/iar/aws_demos`, choose **aws_demos.eww**, and then choose **OK**.
3. Right-click the project name (`aws_demos`), and then choose **Make**.

To run the FreeRTOS demo in IAR Embedded Workbench

1. Make sure the Sense On Power (SOP) jumper on your Texas Instruments CC3220SF-LAUNCHXL is in position 0. For more information, see SimpleLink Wi-Fi CC3x20, CC3x3x Network Processor User's Guide.

2. Use a USB cable to connect your Texas Instruments CC3220SF-LAUNCHXL to your computer.
3. Rebuild your project.

   To rebuild the project, from the **Project** menu, choose **Make**.
4. From the **Project** menu, choose **Download and Debug**. You can ignore "Warning: Failed to initialize EnergyTrace," if it's displayed. For more information about EnergyTrace, see MSP EnergyTrace Technology.
5. When the debugger stops at the breakpoint in `main()`, go to the **Debug** menu, and choose **Go**.

Using CMake with FreeRTOS

If you prefer not to use an IDE for FreeRTOS development, you can alternatively use CMake to build and run the demo applications or applications that you have developed using third-party code editors and debugging tools.

To build the FreeRTOS demo with CMake

1. Create a folder to contain the generated build files (*`build-folder`*).
2. Make sure your search path (**$PATH** environment variable) contains the folder where the TI CGT compiler binary is located (for example `C:\ti\ccs910\ccs\tools\compiler\ti-cgt-arm_18.12.2.LTS\bin`).

   If you are using the TI ARM compiler with your TI board, use the following command to generate build files from source code:

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

For more information, see Using CMake with FreeRTOS.

Troubleshooting

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

To configure debug settings for TI boards

1. In Code Composer, on **Project Explorer**, choose **aws_demos**.
2. From the **Run** menu, choose **Debug Configurations**.
3. In the navigation pane, choose **aws_demos**.
4. On the **Target** tab, under **Connection Options**, choose **Reset the target on a connect**.
5. Choose **Apply**, and then choose **Close**.

If these steps don't work, look at the program's output in the serial terminal. You should see some text that indicates the source of the problem.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Windows Device Simulator

This tutorial provides instructions for getting started with the FreeRTOS Windows Device Simulator.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

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

> **Note**
>
> This tutorial cannot be successfully run on Amazon EC2 Windows instances.

Set up your development environment

1. Install the latest version of Npcap. Select the "WinPcap API-compatible mode" during installation.
2. Install Microsoft Visual Studio.

Visual Studio versions 2017 and 2019 are known to work. All editions of these Visual Studio versions are supported (Community, Professional, or Enterprise).

In addition to the IDE, install the **Desktop development with C++** component.

Install the latest Windows 10 SDK. You can choose this under the **Optional** section of the **Desktop development with C++** component.

3. Make sure that you have an active hard-wired Ethernet connection.
4. (Optional) If you would like to use the CMake-based build system to build your FreeRTOS projects, install the latest version of CMake. FreeRTOS requires CMake version 3.13 or later.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

When the demo project successfully runs on your device you see "Hello World!" sent multiple times to the topic that you subscribed to.

Build and run the FreeRTOS demo project

You can use Visual Studio or CMake to build FreeRTOS projects.

Building and running the FreeRTOS demo project with the Visual Studio IDE

1. Load the project into Visual Studio.

   In Visual Studio, from the **File** menu, choose **Open**. Choose **File/Solution**, navigate to the `projects/pc/windows/visual_studio/aws_demos/aws_demos.sln` file, and then choose **Open**.
2. Retarget the demo project.

The provided demo project depends on the Windows SDK, but it does not have a Windows SDK version specified. By default, the IDE might attempt to build the demo with an SDK version not present on your machine. To set the Windows SDK version, right-click on `aws_demos` and then choose **Retarget Projects**. This opens the **Review Solution Actions** window. Choose a Windows SDK version that is present on your machine (the initial value in the dropdown is fine), and then choose **OK**.

3. Build and run the project.

   From the **Build** menu, choose **Build Solution**, and make sure the solution builds without errors or warnings. Choose **Debug**, **Start Debugging** to run the project. On the first run, you must select a network interface.

Building and running the FreeRTOS demo project with CMake

We recommend that you use the CMake GUI instead of the CMake command line tool to build the demo project for the Windows Simulator.

After you install CMake, open the CMake GUI. On Windows, you can find this from the Start menu under **CMake**, **CMake (cmake-gui)**.

1. Set the FreeRTOS source code directory.

   In the GUI, set the FreeRTOS source code directory (*`freertos`*) for **Where is the source code**.

   Set *`freertos`*`/build` for **Where to build the binaries**.

2. Configure the CMake Project.

   In the CMake GUI, choose **Add Entry**, and on the **Add Cache Entry** window, set the following values:

   Name

   AFR_BOARD

   Type

   STRING

   Value

   pc.windows

Description

(Optional)

3. Choose **Configure**. If CMake prompts you to create the build directory, choose **Yes**, and then select a generator under **Specify the generator for this project**. We recommend using Visual Studio as the generator, but Ninja is also supported. (Note that when using Visual Studio 2019, the platform should be set to Win32 instead of its default setting.) Keep the other generator options unchanged and choose **Finish**.
4. Generate and Open the CMake Project.

   After you have configured the project, the CMake GUI shows all options available for the generated project. For the purposes of this tutorial, you can leave the options at their default values.

   Choose **Generate** to create a Visual Studio solution, and then choose **Open Project** to open the project in Visual Studio.

   In Visual Studio, right-click the `aws_demos` project and choose **Set as StartUp Project**. This enables you to build and run the project. On the first run, you must select a network interface.

For more information about using CMake with FreeRTOS, see Using CMake with FreeRTOS.

Configure your network interface

On the first run of the demo project, you must select the network interface to use. The program counts your network interfaces. Find the number for your hard-wired Ethernet interface. The output should look like this:

```
0 0 [None] FreeRTOS_IPInit
1 0 [None] vTaskStartScheduler
1. rpcap://\Device\NPF_{AD01B877-A0C1-4F33-8256-EE1F4480B70D}
(Network adapter 'Intel(R) Ethernet Connection (4) I219-LM' on local host)

2. rpcap://\Device\NPF_{337F7AF9-2520-4667-8EFF-2B575A98B580}
(Network adapter 'Microsoft' on local host)

The interface that will be opened is set by "configNETWORK_INTERFACE_TO_USE", which
should be defined in FreeRTOSConfig.h

ERROR:  configNETWORK_INTERFACE_TO_USE is set to 0, which is an invalid value.
```

```
Please set configNETWORK_INTERFACE_TO_USE to one of the interface numbers listed above,
then re-compile and re-start the application.  Only Ethernet (as opposed to Wi-Fi)
interfaces are supported.
```

After you have identified the number for your hard-wired Ethernet interface, close the application window. In the previous example, the number to use is 1.

Open `FreeRTOSConfig.h` and set `configNETWORK_INTERFACE_TO_USE` to the number that corresponds to your hard-wired network interface.

> **Important**
>
> Only Ethernet interfaces are supported. Wi-Fi isn't supported.

Troubleshooting

Troubleshooting common problems on Windows

You might run into the following error when trying to build the demo project with Visual Studio:

```
Error "The Windows SDK version X.Y was not found" when building the provided Visual
 Studio solution.
```

The project must be targeted to a Windows SDK version present on your computer.

For general troubleshooting information about getting started with FreeRTOS, see Troubleshooting getting started.

Getting started with the Xilinx Avnet MicroZed Industrial IoT Kit

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial provides instructions for getting started with the Xilinx Avnet MicroZed Industrial IoT Kit. If you do not have the Xilinx Avnet MicroZed Industrial IoT Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

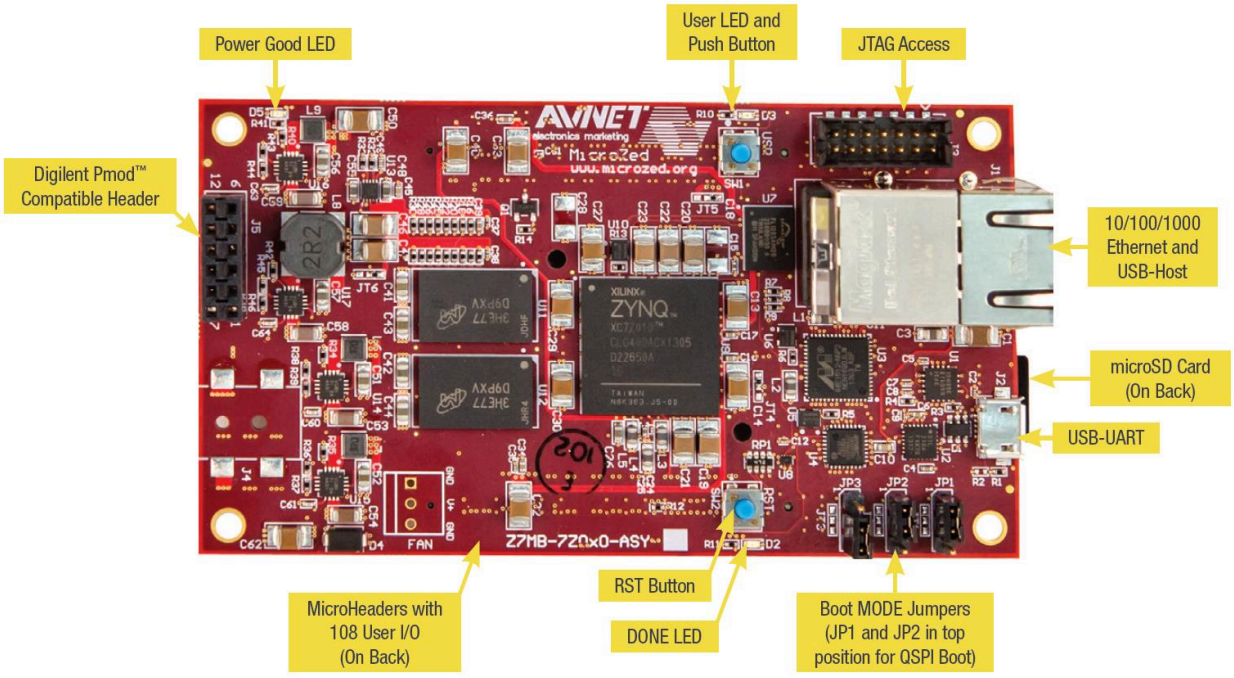
Overview

This tutorial contains instructions for the following getting started steps:

1. Connecting your board to a host machine.
2. Installing software on the host machine for developing and debugging embedded applications for your microcontroller board.
3. Cross compiling a FreeRTOS demo application to a binary image.
4. Loading the application binary image to your board, and then running the application.

Set up the MicroZed hardware

The following diagram might be helpful when you set up the MicroZed hardware:



To set up the MicroZed board

1. Connect your computer to the USB-UART port on your MicroZed board.
2. Connect your computer to the JTAG Access port on your MicroZed board.
3. Connect a router or internet-connected Ethernet port to the Ethernet and USB-Host port on your MicroZed board.

Set up your development environment

To set up FreeRTOS configurations for the MicroZed kit, you must use the Xilinx Software Development Kit (XSDK). XSDK is supported on Windows and Linux.

Download and install XSDK

To install Xilinx software, you need a free Xilinx account.

To download the XSDK

1. Go to the Software Development Kit Standalone WebInstall Client download page.
2. Choose the option appropriate for your operating system.
3. You are directed to a Xilinx sign-in page.

   If you have an account with Xilinx, enter your sign-in credentials and then choose **Sign in**.

   If you do not have an account, choose **Create your account**. After you register, you should receive an email with a link to activate your Xilinx account.
4. On the **Name and Address Verification** page, enter your information and then choose **Next**. The download should be ready to start.
5. Save the `Xilinx_SDK_`*`version_os`* file.

To install the XSDK

1. Open the `Xilinx_SDK_`*`version_os`* file.
2. In **Select Edition to Install**, choose **Xilinx Software Development Kit (XSDK)** and then choose **Next**.
3. On the following page of the installation wizard, under **Installation Options**, select **Install Cable Drivers** and then choose **Next**.

If your computer does not detect the MicroZed's USB-UART connection, install the CP210x USB-to-UART Bridge VCP drivers manually. For instructions, see the Silicon Labs CP210x USB-to-UART Installation Guide.

For more information about XSDK, see the Getting Started with Xilinx SDK on the Xilink website.

Monitoring MQTT messages on the cloud

Before you run the FreeRTOS demo project, you can set up the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud.

To subscribe to the MQTT topic with the AWS IoT MQTT client

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***your-thing-name*`/example/topic`**, and then choose **Subscribe to topic**.

Build and run the FreeRTOS demo project

Open the FreeRTOS demo in the XSDK IDE

1. Launch the XSDK IDE with the workspace directory set to *`freertos`*`/projects/xilinx/microzed/xsdk`.
2. Close the welcome page. From the menu, choose **Project**, and then clear **Build Automatically**.
3. From the menu, choose **File**, and then choose **Import**.
4. On the **Select** page, expand **General**, choose **Existing Projects into Workspace**, and then choose **Next**.
5. On the **Import Projects** page, choose **Select root directory**, and then enter the root directory of your demo project: *`freertos`*`/projects/xilinx/microzed/xsdk/aws_demos`. To browse for the directory, choose **Browse**.

   After you specify a root directory, the projects in that directory appear on the **Import Projects** page. All available projects are selected by default.

> **Note**
>
> If you see a warning at the top of the **Import Projects** page ("Some projects cannot be imported because they already exist in the workspace.") you can ignore it.

6. With all of the projects selected, choose **Finish**.
7. If you don't see the `aws_bsp`, `fsbl`, and `MicroZed_hw_platform_0` projects in the projects pane, repeat the previous steps starting from #3 but with the root directory set to *`freertos`*`/vendors/xilinx`, and import `aws_bsp`, `fsbl`, and `MicroZed_hw_platform_0`.
8. From the menu, choose **Window**, and then choose **Preferences**.
9. In the navigation pane, expand **Run/Debug**, choose **String Substitution**, and then choose **New**.
10. In **New String Substitution Variable**, for **Name**, enter **AFR_ROOT**. For **Value**, enter the root path of the *`freertos`*`/projects/xilinx/microzed/xsdk/aws_demos`. Choose **OK**, and then choose **OK** to save the variable and close **Preferences**.

Build the FreeRTOS demo project

1. In the XSDK IDE, from the menu, choose **Project**, and then choose **Clean**.
2. In **Clean**, leave the options at their default values, and then choose **OK**. XSDK cleans and builds all of the projects, and then generates `.elf` files.

> **Note**
>
> To build all projects without cleaning them, choose **Project**, and then choose **Build All**. To build individual projects, select the project you want to build, choose **Project**, and then choose **Build Project**.

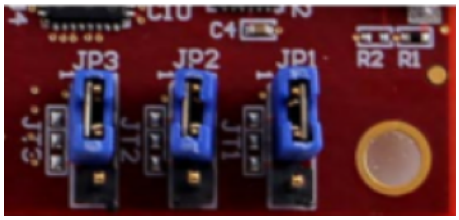
Generate the boot image for the FreeRTOS demo project

1. In the XSDK IDE, right-click **aws_demos**, and then choose **Create Boot Image**.
2. In **Create Boot Image**, choose **Create new BIF file**.
3. Next to **Output BIF file path**, choose **Browse**, and then choose `aws_demos.bif` located at *`<freertos>`*`/vendors/xilinx/microzed/aws_demos/aws_demos.bif`.

4. Choose **Add**.
5. On **Add new boot image partition**, next to **File path**, choose **Browse**, and then choose `fsbl.elf`, located at `vendors/xilinx/fsbl/Debug/fsbl.elf`.
6. For the **Partition type**, choose **bootloader**, and then choose **OK**.
7. On **Create Boot Image**, choose **Create Image**. On **Override Files**, choose **OK** to overwrite the existing `aws_demos.bif` and generate the `BOOT.bin` file at `projects/xilinx/microzed/xsdk/aws_demos/BOOT.bin`.

JTAG debugging

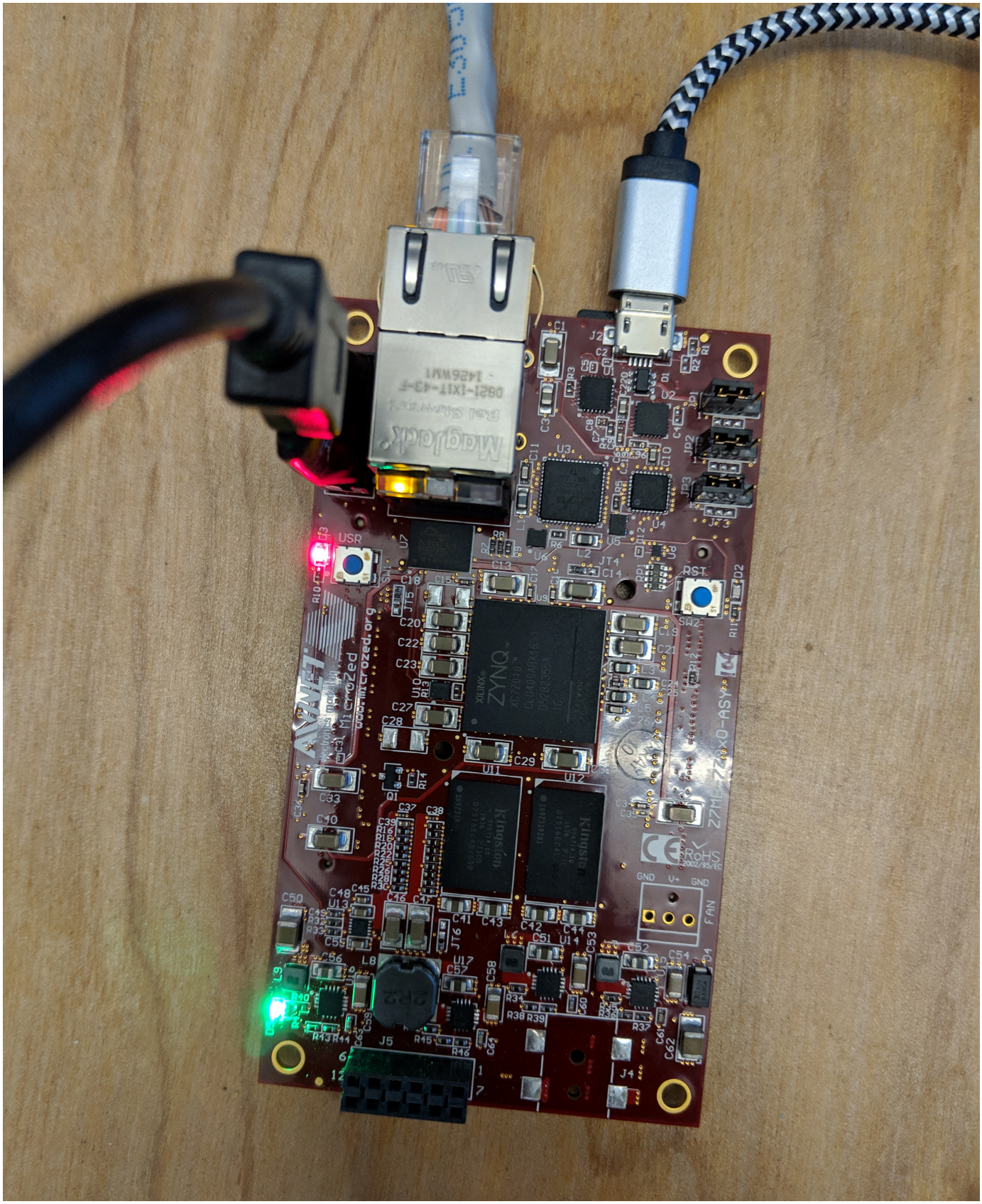
1. Set your MicroZed board's boot mode jumpers to the JTAG boot mode.



2. Insert your MicroSD card into the MicroSD card slot located directly under the USB-UART port.

> **Note**
>
> Before you debug, be sure to back up any content that you have on the MicroSD card.

Your board should look similar to the following:


USR
RST
ZYNQ
Kingston
Kingston
RoHS
GND V+ GND
FAN

3. In the XSDK IDE, right-click **aws_demos**, choose **Debug As**, and then choose **1 Launch on System Hardware (System Debugger)**.
4. When the debugger stops at the breakpoint in `main()`, from the menu, choose **Run**, and then choose **Resume**.

> **Note**
>
> The first time you run the application, a new certificate-key pair is imported into non-volatile memory. For subsequent runs, you can comment out `vDevModeKeyProvisioning()` in the `main.c` file before you rebuild the images and the `BOOT.bin` file. This prevents the copying of the certificates and key to storage on every run.

You can opt to boot your MicroZed board from a MicroSD card or from QSPI flash to run the FreeRTOS demo project. For instructions, see Generate the boot image for the FreeRTOS demo project and Run the FreeRTOS demo project.

Run the FreeRTOS demo project

To run the FreeRTOS demo project, you can boot your MicroZed board from a MicroSD card or from QSPI flash.

As you set up your MicroZed board for running the FreeRTOS demo project, refer to the diagram in Set up the MicroZed hardware. Make sure that you have connected your MicroZed board to your computer.

Boot the FreeRTOS project from a MicroSD card

Format the MicroSD card that is provided with the Xilinx MicroZed Industrial IoT Kit.

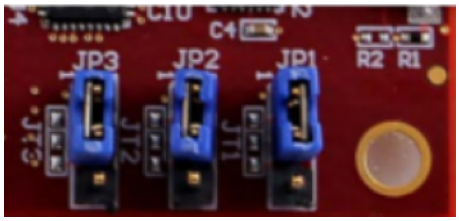
1. Copy the `BOOT.bin` file to the MicroSD card.
2. Insert the card into the MicroSD card slot directly under the USB-UART port.
3. Set the MicroZed boot mode jumpers to SD boot mode.



4. Press the RST button to reset the device and start booting the application. You can also unplug the USB-UART cable from the USB-UART port, and then reinsert the cable.

Boot the FreeRTOS demo project from QSPI flash

1. Set your MicroZed board's boot mode jumpers to the JTAG boot mode.



2. Verify that your computer is connected to the USB-UART and JTAG Access ports. The green Power Good LED light should be illuminated.
3. In the XSDK IDE, from the menu, choose **Xilinx**, and then choose **Program Flash**.
4. In **Program Flash Memory**, the hardware platform should be filled in automatically. For **Connection**, choose your MicroZed hardware server to connect your board with your host computer.

> **Note**
>
> If you are using the Xilinx Smart Lync JTAG cable, you must create a hardware server in XSDK IDE. Choose **New**, and then define your server.

5. In **Image File**, enter the directory path to your `BOOT.bin` image file. Choose **Browse** to browse for the file instead.
6. In **Offset**, enter **`0x0`**.
7. In **FSBL File**, enter the directory path to your `fsbl.elf` file. Choose **Browse** to browse for the file instead.
8. Choose **Program** to program your board.
9. After the QSPI programming is complete, remove the USB-UART cable to power off the board.
10. Set your MicroZed board's boot mode jumpers to the QSPI boot mode.
11. Insert your card into the MicroSD card slot located directly under the USB-UART port.

> **Note**
>
> Be sure to back up any content that you have on the MicroSD card.

12. Press the RST button to reset the device and start booting the application. You can also unplug the USB-UART cable from the USB-UART port, and then reinsert the cable.

Troubleshooting

If you encounter build errors that are related to incorrect paths, try to clean and rebuild the project, as described in Build the FreeRTOS demo project.

If you are using Windows, make sure that you use forward slashes when you set the string substitution variables in the Windows XSDK IDE.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started.

Next steps with FreeRTOS

> **Important**
>
> This page refers to the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

After you build, flash, and run the FreeRTOS demo project for your board, you can visit the FreeRTOS.org website to learn more about Creating a New FreeRTOS Project. There are also demos for many FreeRTOS libraries that show how to perform important tasks, interact with AWS IoT services, and program board-specific capabilities (such as cellular modems). For more information, see the FreeRTOS Library Categories page.

The FreeRTOS.org website also has in-depth information on the FreeRTOS Kernel as well as fundamental real-time operating system concepts. For more information, see the FreeRTOS Kernel Developer Docs and FreeRTOS Kernel Secondary Docs pages.

FreeRTOS Over-the-Air Updates

> **Note**
>
> See AWS IoT Over-the-Air (OTA) on the FreeRTOS website for recent information on performing Over-the-air (OTA) updates.

Over-the-air (OTA) updates allow you to deploy firmware updates to one or more devices in your fleet. Although OTA updates were designed to update device firmware, you can use them to send any files to one or more devices registered with AWS IoT. When you send updates over the air, we recommend that you digitally sign them so that the devices that receive the files can verify they haven't been tampered with en route.

You can use Code Signing for AWS IoT to sign your files, or you can sign your files with your own code-signing tools.

When you create an OTA update, the OTA Update Manager service creates an AWS IoT job to notify your devices that an update is available. The OTA demo application runs on your device and creates a FreeRTOS task that subscribes to notification topics for AWS IoT jobs and listens for update messages. When an update is available, the OTA Agent publishes requests to AWS IoT and receives updates using the HTTP or MQTT protocol, depending on the settings you chose. The OTA Agent checks the digital signature of the downloaded files and, if the files are valid, installs the firmware update. If you don't use the FreeRTOS OTA Update demo application, you must integrate the AWS IoT Over the air (OTA) library into your own application to get the firmware update capability.

FreeRTOS over-the-air updates make it possible for you to:

- Digitally sign firmware before deployment.
- Deploy new firmware images to a single device, a group of devices, or your entire fleet.
- Deploy firmware to devices as they are added to groups, reset, or reprovisioned.
- Verify the authenticity and integrity of new firmware after it's deployed to devices.
- Monitor the progress of a deployment.
- Debug a failed deployment.

Tagging OTA resources

To help you manage your OTA resources, you can optionally assign your own metadata to updates and streams in the form of tags. Tags make it possible for you to categorize your AWS IoT resources in different ways (for example, by purpose, owner, or environment). This is useful when you have many resources of the same type. You can quickly identify a resource based on the tags you've assigned to it.

For more information, see Tagging Your AWS IoT Resources.

OTA update prerequisites

To use over-the-air (OTA) updates, do the following:

- Check the Prerequisites for OTA updates using HTTP or the Prerequisites for OTA updates using MQTT.
- Create an Amazon S3 bucket to store your update.
- Create an OTA Update service role.
- Create an OTA user policy.
- Create a code-signing certificate.
- If you are using Code Signing for AWS IoT, Grant access to code signing for AWS IoT.
- Download FreeRTOS with the OTA library.

Create an Amazon S3 bucket to store your update

OTA update files are stored in Amazon S3 buckets.

If you're using Code Signing for AWS IoT, the command that you use to create a code-signing job takes a source bucket (where the unsigned firmware image is located) and a destination bucket (where the signed firmware image is written). You can specify the same bucket for the source and destination. The file names are changed to GUIDs so the original files are not overwritten.

To create an Amazon S3 bucket

1. Sign in to the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. Choose **Create bucket**.
3. Enter a **bucket name**.

4. Under **Bucket settings for Block Public Access** keep **Block all public access** selected to accept the default permissions.
5. Under **Bucket Versioning**, select **Enable** to keep all versions in the same bucket.
6. Choose **Create bucket**.

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

Create an OTA Update service role

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

To create an OTA service role

1. Sign in to the https://console.aws.amazon.com/iam/.
2. From the navigation pane, choose **Roles**.
3. Choose **Create role**.
4. Under **Select type of trusted entity**, choose **AWS Service**.
5. Choose **IoT** from the list of AWS services.
6. Under **Select your use case**, choose **IoT**.
7. Choose **Next: Permissions**.
8. Choose **Next: Tags**.
9. Choose **Next: Review**.
10. Enter a role name and description, and then choose **Create role**.

For more information about IAM roles, see IAM Roles.

> **Important**
>
> To address the confused deputy security issue, you must follow instructions in the AWS IoT Core guide.

To add OTA update permissions to your OTA service role

1. In the search box on the IAM console page, enter the name of your role, and then choose it from the list.

2. Choose **Attach policies**.
3. In the **Search** box, enter "AmazonFreeRTOSOTAUpdate", select **AmazonFreeRTOSOTAUpdate** from the list of filtered policies, and then choose **Attach policy** to attach the policy to your service role.

To add the required IAM permissions to your OTA service role

1. In the search box on the IAM console page, enter the name of your role, and then choose it from the list.
2. Choose **Add inline policy**.
3. Choose the **JSON** tab.
4. Copy and paste the following policy document into the text box:

   JSON

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::111122223333:role/your_role_name"
        }
    ]
}
```

   Make sure that you replace *your_account_id* with your AWS account ID, and *your_role_name* with the name of the OTA service role.
5. Choose **Review policy**.
6. Enter a name for the policy, and then choose **Create policy**.

> **Note**
>
> The following procedure isn't required if your Amazon S3 bucket name begins with "afr-ota". If it does, the AWS managed policy `AmazonFreeRTOSOTAUpdate` already includes the required permissions.

To add the required Amazon S3 permissions to your OTA service role

1. In the search box on the IAM console page, enter the name of your role, and then choose it from the list.
2. Choose **Add inline policy**.
3. Choose the **JSON** tab.
4. Copy and paste the following policy document into the box.

   JSON

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

   This policy grants your OTA service role permission to read Amazon S3 objects. Make sure that you replace *example-bucket* with the name of your bucket.

5. Choose **Review policy**.

6. Enter a name for the policy, and then choose **Create policy**.

Create an OTA user policy

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

- Access the S3 bucket where your firmware updates are stored.
- Access certificates stored in AWS Certificate Manager.
- Access the AWS IoT MQTT-based file delivery feature.
- Access FreeRTOS OTA updates.
- Access AWS IoT jobs.
- Access IAM.
- Access Code Signing for AWS IoT. See Grant access to code signing for AWS IoT.
- List FreeRTOS hardware platforms.
- Tag and untag AWS IoT resources.

To grant your user the required permissions, see IAM Policies. Also see Authorizing users and cloud services to use AWS IoT Jobs.

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Create a code-signing certificate

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

Different platforms require different types of code-signing certificates. The following sections describe how to create code-signing certificates for different FreeRTOS-qualified platforms.

Topics

- Creating a code-signing certificate for the Texas Instruments CC3220SF-LAUNCHXL
- Creating a code-signing certificate for the Espressif ESP32
- Creating a code-signing certificate for the Nordic nrf52840-dk
- Creating a code-signing certificate for the FreeRTOS Windows simulator
- Creating a code-signing certificate for custom hardware

Creating a code-signing certificate for the Texas Instruments CC3220SF-LAUNCHXL

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

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

- Production (certificate-catalog)

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

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

Use the AWS Command Line Interface to import your code-signing certificate, private key, and certificate chain into AWS Certificate Manager. For more information see Installing the AWS CLI in the *AWS Command Line Interface User Guide*.

Download and install the latest version of SimpleLink CC3220 SDK. By default, the files you need are located here:

`C:\ti\simplelink_cc32xx_sdk_version\tools\cc32xx_tools\certificate-playground` (Windows)

`/Applications/Ti/simplelink_cc32xx_version/tools/cc32xx_tools/certificate-playground` (macOS)

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

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

> **Note**
>
> To create a code signing certificate, install OpenSSL on your machine. After you install OpenSSL, make sure that `openssl` is assigned to the OpenSSL executable in your command prompt or terminal environment.

To create a self-signed code signing certificate

1. Open a command prompt or terminal with administrator permissions.
2. In your working directory, use the following text to create a file named `cert_config.txt`. Replace *`test_signer@amazon.com`* with your email address.

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

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

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

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

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

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

   The TI playground root CA private key is located here:

   `C:\ti\simplelink_cc32xx_sdk_`*`version`*`\tools\cc32xx_tools\certificate-playground\dummy-root-ca-cert-key` (Windows)

   `/Applications/Ti/simplelink_cc32xx_sdk_`*`version`*`/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert-key` (macOS)

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

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

   The TI playground root certificate is located here:

   `C:\ti\simplelink_cc32xx_sdk_`*`version`*`\tools\cc32xx_tools\certificate-playground/dummy-root-ca-cert` (Windows)

   `/Applications/Ti/simplelink_cc32xx_sdk_`*`version`*`/tools/cc32xx_tools/certificate-playground/dummy-root-ca-cert` (macOS)

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

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

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

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

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

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

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

```
aws acm import-certificate --certificate fileb://tisigner.crt.pem --private-key
 fileb://tisigner.key --certificate-chain fileb://dummy-root-ca-cert.pem
```

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

> **Note**
>
> This step is written with the assumption that you are going to use Code Signing for AWS IoT to sign your firmware images. Although the use of Code Signing for AWS IoT is recommended, you can sign your firmware images manually.

Creating a code-signing certificate for the Espressif ESP32

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The Espressif ESP32 boards support a self-signed SHA-256 with ECDSA code-signing certificate.

Note

To create a code signing certificate, install OpenSSL on your machine. After you install OpenSSL, make sure that `openssl` is assigned to the OpenSSL executable in your command prompt or terminal environment.
Use the AWS Command Line Interface to import your code-signing certificate, private key, and certificate chain into AWS Certificate Manager. For information about installing the AWS CLI, see Installing the AWS CLI.

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

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

[ my_dn ]
commonName = test_signer@amazon.com

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

2. Create an ECDSA code-signing private key:

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
 ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. Create an ECDSA code-signing certificate:

```
openssl req -new -x509 -config cert_config.txt -extensions my_exts -nodes -days 365
 -key ecdsasigner.key -out ecdsasigner.crt
```

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

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

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

> **Note**
>
> This step is written with the assumption that you are going to use Code Signing for AWS IoT to sign your firmware images. Although the use of Code Signing for AWS IoT is recommended, you can sign your firmware images manually.

Creating a code-signing certificate for the Nordic nrf52840-dk

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The Nordic nrf52840-dk supports a self-signed SHA256 with ECDSA code-signing certificate.

> **Note**
>
> To create a code signing certificate, install OpenSSL on your machine. After you install OpenSSL, make sure that `openssl` is assigned to the OpenSSL executable in your command prompt or terminal environment.
> Use the AWS Command Line Interface to import your code-signing certificate, private key, and certificate chain into AWS Certificate Manager. For information about installing the AWS CLI, see Installing the AWS CLI.

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

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

```
[ my_dn ]
commonName = test_signer@amazon.com

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

2. Create an ECDSA code-signing private key:

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
 ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. Create an ECDSA code-signing certificate:

```
openssl req -new -x509 -config cert_config.txt -extensions my_exts -nodes -days 365
 -key ecdsasigner.key -out ecdsasigner.crt
```

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

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

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

> **Note**
>
> This step is written with the assumption that you are going to use Code Signing for AWS IoT to sign your firmware images. Although the use of Code Signing for AWS IoT is recommended, you can sign your firmware images manually.

Creating a code-signing certificate for the FreeRTOS Windows simulator

The FreeRTOS Windows simulator requires a code-signing certificate with an ECDSA P-256 key and SHA-256 hash to perform OTA updates. If you don't have a code-signing certificate, follow these steps to create one.

Note

To create a code-signing certificate, install OpenSSL on your machine. After you install OpenSSL, make sure that `openssl` is assigned to the OpenSSL executable in your command prompt or terminal environment.
Use the AWS Command Line Interface to import your code-signing certificate, private key, and certificate chain into AWS Certificate Manager. For information about installing the AWS CLI, see Installing the AWS CLI.

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

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

[ my_dn ]
commonName = test_signer@amazon.com

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

2. Create an ECDSA code-signing private key:

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
 ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. Create an ECDSA code-signing certificate:

```
openssl req -new -x509 -config cert_config.txt -extensions my_exts -nodes -days 365
 -key ecdsasigner.key -out ecdsasigner.crt
```

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

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

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

> **Note**
>
> This step is written with the assumption that you are going to use Code Signing for AWS IoT to sign your firmware images. Although the use of Code Signing for AWS IoT is recommended, you can sign your firmware images manually.

Creating a code-signing certificate for custom hardware

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

Use the AWS Command Line Interface to import your code-signing certificate, private key, and certificate chain into AWS Certificate Manager. For information about installing the AWS CLI, see Installing the AWS CLI.

After you create your code-signing certificate, you can use the AWS CLI to import it into ACM:

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

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

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

You must copy, paste, and format the contents of your code-signing certificate into the `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/ota_demo_config.h` file that is part of the FreeRTOS code you download later.

Grant access to code signing for AWS IoT

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Download FreeRTOS with the OTA library

You can clone or download FreeRTOS from GitHub. See the README.md file for instructions.

For information about setting up and running the OTA demo application, see Over-the-air updates demo application.

Important

- In this topic, the path to the FreeRTOS download directory is referred to as `freertos`.
- Space characters in the `freertos` path can cause build failures. When you clone or copy the repository, make sure the path you that create doesn't contain space characters.
- The maximum length of a file path on Microsoft Windows is 260 characters. Long FreeRTOS download directory paths can cause build failures.
- Because the source code may contain symbolic links, if you're using Windows to extract the archive, you may have to:
  - Enable Developer Mode or,
  - Use a console that is elevated as administrator.

  In this way, Windows can properly create symbolic links when it extracts the archive. Otherwise, symbolic links will be written as normal files that contain the paths of the symbolic links as text or are empty. For more information, see the blog entry Symlinks in Windows 10!.

If you use Git under Windows, you must enable Developer Mode or you must:

- Set `core.symlinks` to true with the following command:

```
git config --global core.symlinks true
```

- Use a console that is elevated as administrator whenever you use a git command that writes to the system (for example, **git pull**, **git clone**, and **git submodule update --init --recursive**).

Prerequisites for OTA updates using MQTT

This section describes the general requirements for using MQTT to perform over-the-air (OTA updates).

Minimum requirements

- Device firmware must include the necessary FreeRTOS libraries (coreMQTT Agent, OTA update, and their dependencies).
- FreeRTOS version 1.4.0 or later is required. However, we recommend that you use the latest version when possible.

Configurations

Beginning with version 201912.00, FreeRTOS OTA can use either the HTTP or MQTT protocol to transfer firmware update images from AWS IoT to devices. If you specify both protocols when you create an OTA update in FreeRTOS, each device will determine the protocol used to transfer the image. See Prerequisites for OTA updates using HTTP for more information.

By default, the configuration of the OTA protocols in `ota_config.h` is to use the MQTT protocol.

Device specific configurations

None.

Memory usage

When MQTT is used for data transfer, no additional memory is required for the MQTT connection because it's shared between control and data operations.

Device policy

Each device that receives an OTA update using MQTT must be registered as a thing in AWS IoT and the thing must have an attached policy like the one listed here. You can find more information about the items in the `"Action"` and `"Resource"` objects at AWS IoT Core Policy Actions and AWS IoT Core Action Resources.

Notes

- The `iot:Connect` permissions allow your device to connect to AWS IoT over MQTT.
- The `iot:Subscribe` and `iot:Publish` permissions on the topics of AWS IoT jobs (`.../jobs/*`) allow the connected device to receive job notifications and job documents, and to publish the completion state of a job execution.
- The `iot:Subscribe` and `iot:Publish` permissions on the topics of AWS IoT OTA streams (`.../streams/*`) allow the connected device to fetch OTA update data from AWS IoT. These permissions are required to perform firmware updates over MQTT.
- The `iot:Receive` permissions allow AWS IoT Core to publish messages on those topics to the connected device. This permission is checked on every delivery of an MQTT message. You can use this permission to revoke access to clients that are currently subscribed to a topic.

Prerequisites for OTA updates using HTTP

This section describes the general requirements for using HTTP to perform over-the-air (OTA) updates. Beginning with version 201912.00, FreeRTOS OTA can use either the HTTP or MQTT protocol to transfer firmware update images from AWS IoT to devices.

> **Note**
>
> - Although the HTTP protocol might be used to transfer the firmware image, the coreMQTT Agent library is still required because other interactions with AWS IoT Core use the coreMQTT Agent library, including sending or receiving job execution notifications, job documents, and execution status updates.
> - When you specify both MQTT and HTTP protocols for the OTA update job, the setup of the OTA Agent software on each individual device determines the protocol used to transfer the firmware image. To change the OTA Agent from the default MQTT protocol

method to the HTTP protocol, you can modify the header files used to compile the FreeRTOS source code for the device.

Minimum requirements

- Device firmware must include the necessary FreeRTOS libraries (coreMQTT Agent, HTTP, OTA Agent, and their dependencies).
- FreeRTOS version 201912.00 or later is required to change the configuration of the OTA protocols to enable OTA data transfer over HTTP.

Configurations

See the following configuration of the OTA protocols in the `\vendors\boards\board\aws_demos\config_files\ota_config.h` file.

To enable OTA data transfer over HTTP

1. Change `configENABLED_DATA_PROTOCOLS` to `OTA_DATA_OVER_HTTP`.
2. When the OTA updates, you can specify both protocols so that either MQTT or HTTP protocol can be used., You can set the primary protocol used by the device to HTTP by changing `configOTA_PRIMARY_DATA_PROTOCOL` to `OTA_DATA_OVER_HTTP`.

Note

HTTP is only supported for OTA data operations. For control operations, you must use MQTT.

Device specific configurations

ESP32

Due to a limited amount of RAM, you must turn off BLE when you enable HTTP as an OTA data protocol. In the `vendors/espressif/boards/esp32/aws_demos/config_files/aws_iot_network_config.h` file, change `configENABLED_NETWORKS` to `AWSIOT_NETWORK_TYPE_WIFI` only.

```
/**
     * @brief Configuration flag which is used to enable one or more network
 interfaces for a board.
     *
     * The configuration can be changed any time to keep one or more network enabled
 or disabled.
     * More than one network interfaces can be enabled by using 'OR' operation with
 flags for
     * each network types supported. Flags for all supported network types can be
 found
     * in "aws_iot_network.h"
     *
     */
    #define configENABLED_NETWORKS      ( AWSIOT_NETWORK_TYPE_WIFI )
```

Memory usage

When MQTT is used for data transfer, no additional heap memory is required for the MQTT connection because it's shared between control and data operations. However, enabling data over HTTP requires additional heap memory. The following is the heap memory usage data for all supported platforms, calculated using the FreeRTOS `xPortGetFreeHeapSize` API. You must make sure there is enough RAM to use the OTA library.

Texas Instruments CC3220SF-LAUNCHXL

Control operations (MQTT): 12 KB

Data operations (HTTP): 10 KB

> **Note**
>
> TI uses significantly less RAM because it does SSL on hardware, so it doesn't use the mbedtls library.

Microchip Curiosity PIC32MZEF

Control operations (MQTT): 65 KB

Data operations (HTTP): 43 KB

Espressif ESP32

Control operations (MQTT): 65 KB

Data operations (HTTP): 45 KB

> **Note**
>
> BLE on ESP32 takes about 87 KB RAM. There's not enough RAM to enable all of them, which is mentioned in the device specific configurations above.

Windows simulator

Control operations (MQTT): 82 KB

Data operations (HTTP): 63 KB

Nordic nrf52840-dk

HTTP is not supported.

Device policy

This policy allows you to use either MQTT or HTTP for OTA updates.

Each device that receives an OTA update using HTTP must be registered as a thing in AWS IoT and the thing must have an attached policy like the one listed here. You can find more information about the items in the `"Action"` and `"Resource"` objects at AWS IoT Core Policy Actions and AWS IoT Core Action Resources.

Notes

- The `iot:Connect` permissions allow your device to connect to AWS IoT over MQTT.
- The `iot:Subscribe` and `iot:Publish` permissions on the topics of AWS IoT jobs (`.../jobs/*`) allow the connected device to receive job notifications and job documents, and to publish the completion state of a job execution.
- The `iot:Receive` permissions allow AWS IoT Core to publish messages on those topics to the current connected device. This permission is checked on every delivery of an MQTT message. You can use this permission to revoke access to clients that are currently subscribed to a topic.

OTA tutorial

This section contains a tutorial for updating firmware on devices running FreeRTOS using OTA updates. In addition to firmware images, you can use an OTA update to send any type of file to a device connected to AWS IoT.

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

To create an OTA update

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

When you create an OTA update, you specify the image delivery protocol (MQTT or HTTP) or specify both to allow the device to choose. The FreeRTOS OTA agent on the device receives the updated firmware image and verifies the digital signature, checksum, and version number of the new image. If the firmware update is verified, the device is reset and, based on application-defined logic, commits the update. If your devices are not running FreeRTOS, you must implement an OTA agent that runs on your devices.

Installing the initial firmware

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

Topics

- Install the initial version of firmware on the Texas Instruments CC3220SF-LAUNCHXL
- Install the initial version of firmware on the Espressif ESP32

- Install the initial version of firmware on the Nordic nRF52840 DK
- Initial firmware on the Windows simulator
- Install the initial version of firmware on a custom board

Install the initial version of firmware on the Texas Instruments CC3220SF-LAUNCHXL

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

These steps are written with the assumption that you have already built the `aws_demos` project, as described in Download, build, flash, and run the FreeRTOS OTA demo on the Texas Instruments CC3220SF-LAUNCHXL.

1. On your Texas Instruments CC3220SF-LAUNCHXL, place the SOP jumper on the middle set of pins (position = 1) and reset the board.
2. Download and install the TI Uniflash tool.
3. Start Uniflash. From the list of configurations, choose **CC3220SF-LAUNCHXL**, and then choose **Start Image Creator**.
4. Choose **New Project**.
5. On the **Start new project** page, enter a name for your project. For **Device Type**, choose **CC3220SF**. For **Device Mode**, choose **Develop**. Choose **Create Project**.
6. Disconnect your terminal emulator.
7. On the right side of the Uniflash application window, choose **Connect**.
8. Under **Advanced**, **Files**, choose **User Files**.
9. In the **File** selector pane, choose the **Add File** icon

   .
10. Browse to the `/Applications/Ti/simplelink_cc32xx_sdk_`*`version`*`/tools/cc32xx_tools/certificate-playground` directory, select `dummy-root-ca-cert`, choose **Open**, and then choose **Write**.

11. In the **File** selector pane, choose the **Add File** icon

    .

12. Browse to the working directory where you created the code-signing certificate and private key, choose `tisigner.crt.der`, choose **Open**, and then choose **Write**.
13. From the **Action** drop-down list, choose **Select MCU Image**, and then choose **Browse** to choose the firmware image to use write to your device (**aws_demos.bin**). This file is located in the *freertos*`/vendors/ti/boards/cc3220_launchpad/aws_demos/ccs/Debug` directory. Choose **Open**.

    a. In the file dialog box, confirm the file name is set to `mcuflashimg.bin`.

    b. Select the **Vendor** check box.

    c. Under **File Token**, type **1952007250**.

    d. Under **Private Key File Name**, choose **Browse**, and then choose `tisigner.key` from the working directory where you created the code-signing certificate and private key.

    e. Under **Certification File Name**, choose `tisigner.crt.der`.

    f. Choose **Write**.

14. In the left pane, under **Files**, choose **Service Pack**.
15. Under **Service Pack File Name**, choose **Browse**, browse to `simplelink_cc32x_sdk_`*version*`/tools/cc32xx_tools/servicepack-cc3x20`, choose `sp_3.7.0.1_2.0.0.0_2.2.0.6.bin`, and then choose **Open**.
16. In the left pane, under **Files**, choose **Trusted Root-Certificate Catalog**.
17. Clear the **Use default Trusted Root-Certificate Catalog** check box.
18. Under **Source File**, choose **Browse**, choose **simplelink_cc32xx_sdk_*version*/tools/cc32xx_tools/certificate-playground/certcatalogPlayGround20160911.lst**, and then choose **Open**.
19. Under **Signature Source File**, choose **Browse**, choose **simplelink_cc32xx_sdk_*version*/tools/cc32xx_tools/certificate-playground/certcatalogPlayGround20160911.lst.signed_3220.bin**, and then choose **Open**.
20. Choose the

    button to save your project.

21. Choose the

    button.

22. Choose **Program Image (Create and Program)**.

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

    The terminal should display output like the following.

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

Device came up in Station mode

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

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

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

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

[NETAPP EVENT] IP acquired by the device


Device has connected to Guest

Device IP Address is 111.222.3.44


6 1716 [OTA] OTA demo version 0.9.0
7 1717 [OTA] Creating MQTT Client...
8 1717 [OTA] Connecting to broker...
```

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

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

27 2814 [OTA Task] [OTA] Check For Update #0
28 2814 [OTA Task] Sending command to MQTT task.
29 2814 [MQTT] Received message 40000 from queue.
30 2916 [MQTT] MQTT Publish was successful.
31 2916 [MQTT] Notifying task.
32 2917 [OTA Task] Command sent to MQTT task passed.
33 2917 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:TI-LaunchPad ]
34 2917 [OTA Task] [OTA] Missing job parameter: execution
35 2917 [OTA Task] [OTA] Missing job parameter: jobId
36 2918 [OTA Task] [OTA] Missing job parameter: jobDocument
37 2918 [OTA Task] [OTA] Missing job parameter: ts_ota
38 2918 [OTA Task] [OTA] Missing job parameter: files
39 2918 [OTA Task] [OTA] Missing job parameter: streamname
40 2918 [OTA Task] [OTA] Missing job parameter: certfile
41 2918 [OTA Task] [OTA] Missing job parameter: filepath
42 2918 [OTA Task] [OTA] Missing job parameter: filesize
43 2919 [OTA Task] [OTA] Missing job parameter: sig-sha1-rsa
44 2919 [OTA Task] [OTA] Missing job parameter: fileid
45 2919 [OTA Task] [OTA] Missing job parameter: attr
47 3919 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
48 4919 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
```

```
49 5919 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
```

Install the initial version of firmware on the Espressif ESP32

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This guide is written with the assumption that you have already performed the steps in Getting Started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT and Over-the-Air Update Prerequisites. Before you attempt an OTA update, you might want to run the MQTT demo project described in Getting Started with FreeRTOS to ensure that your board and tool chain are set up correctly.

To flash an initial factory image to the board

1. Open *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h, comment out #define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED, and define CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED or CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED.
2. Copy your SHA-256/ECDSA PEM-formatted code-signing certificate that you generated in the OTA update prerequisites to vendors/*vendor*/boards/*board*/aws_demos/config_files/ota_demo_config.h. It should be formatted in following way.

   ```
   #define otapalconfigCODE_SIGNING_CERTIFICATE  \
   "-----BEGIN CERTIFICATE-----\n" \
   "...base64 data...\n" \
   "-----END CERTIFICATE-----\n";
   ```
3. With the OTA Update demo selected, follow the same steps outlined in Getting Started with ESP32 to build and flash the image. If you have previously built and flashed the project, you might need to run make clean first. After you run make flash monitor, you should see

something like the following. The ordering of some messages might vary, because the demo application runs multiple tasks at once.

```
I (28) boot: ESP-IDF v3.1-dev-322-gf307f41-dirty 2nd stage bootloader
I (28) boot: compile time 16:32:33
I (29) boot: Enabling RNG early entropy source...
I (34) boot: SPI Speed : 40MHz
I (38) boot: SPI Mode : DIO
I (42) boot: SPI Flash Size : 4MB
I (46) boot: Partition Table:
I (50) boot: ## Label Usage Type ST Offset Length
I (57) boot: 0 nvs WiFi data 01 02 00010000 00006000
I (64) boot: 1 otadata OTA data 01 00 00016000 00002000
I (72) boot: 2 phy_init RF data 01 01 00018000 00001000
I (79) boot: 3 ota_0 OTA app 00 10 00020000 00100000
I (87) boot: 4 ota_1 OTA app 00 11 00120000 00100000
I (94) boot: 5 storage Unknown data 01 82 00220000 00010000
I (102) boot: End of partition table
I (106) esp_image: segment 0: paddr=0x00020020 vaddr=0x3f400020 size=0x14784
 ( 83844) map
I (144) esp_image: segment 1: paddr=0x000347ac vaddr=0x3ffb0000 size=0x023ec
 ( 9196) load
I (148) esp_image: segment 2: paddr=0x00036ba0 vaddr=0x40080000 size=0x00400
 ( 1024) load
I (151) esp_image: segment 3: paddr=0x00036fa8 vaddr=0x40080400 size=0x09068
 ( 36968) load
I (175) esp_image: segment 4: paddr=0x00040018 vaddr=0x400d0018 size=0x719b8
 (465336) map
I (337) esp_image: segment 5: paddr=0x000b19d8 vaddr=0x40089468 size=0x04934
 ( 18740) load
I (345) esp_image: segment 6: paddr=0x000b6314 vaddr=0x400c0000 size=0x00000 ( 0)
 load
I (353) boot: Loaded app from partition at offset 0x20000
I (353) boot: ota rollback check done
I (354) boot: Disabling RNG early entropy source...
I (360) cpu_start: Pro cpu up.
I (363) cpu_start: Single core mode
I (368) heap_init: Initializing. RAM available for dynamic allocation:
I (375) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM
I (381) heap_init: At 3FFC0748 len 0001F8B8 (126 KiB): DRAM
I (387) heap_init: At 3FFE0440 len 00003BC0 (14 KiB): D/IRAM
I (393) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (400) heap_init: At 4008DD9C len 00012264 (72 KiB): IRAM
```

```
I (406) cpu_start: Pro cpu start user code
I (88) cpu_start: Starting scheduler on PRO CPU.
I (113) wifi: wifi firmware version: f79168c
I (113) wifi: config NVS flash: enabled
I (113) wifi: config nano formating: disabled
I (113) system_api: Base MAC address is not set, read default base MAC address from
 BLK0 of EFUSE
I (123) system_api: Base MAC address is not set, read default base MAC address from
 BLK0 of EFUSE
I (133) wifi: Init dynamic tx buffer num: 32
I (143) wifi: Init data frame dynamic rx buffer num: 32
I (143) wifi: Init management frame dynamic rx buffer num: 32
I (143) wifi: wifi driver task: 3ffc73ec, prio:23, stack:4096
I (153) wifi: Init static rx buffer num: 10
I (153) wifi: Init dynamic rx buffer num: 32
I (163) wifi: wifi power manager task: 0x3ffcc028 prio: 21 stack: 2560
0 6 [main] WiFi module initialized. Connecting to AP <Your_WiFi_SSID>...
I (233) phy: phy_version: 383.0, 79a622c, Jan 30 2018, 15:38:06, 0, 0
I (233) wifi: mode : sta (30:ae:a4:80:0a:04)
I (233) WIFI: SYSTEM_EVENT_STA_START
I (363) wifi: n:1 0, o:1 0, ap:255 255, sta:1 0, prof:1
I (1343) wifi: state: init -> auth (b0)
I (1343) wifi: state: auth -> assoc (0)
I (1353) wifi: state: assoc -> run (10)
I (1373) wifi: connected with <Your_WiFi_SSID>, channel 1
I (1373) WIFI: SYSTEM_EVENT_STA_CONNECTED
1 302 [IP-task] vDHCPProcess: offer c0a86c13ip
I (3123) event: sta ip: 192.168.108.19, mask: 255.255.224.0, gw: 192.168.96.1
I (3123) WIFI: SYSTEM_EVENT_STA_GOT_IP
2 302 [IP-task] vDHCPProcess: offer c0a86c13ip
3 303 [main] WiFi Connected to AP. Creating tasks which use network...
4 304 [OTA] OTA demo version 0.9.6
5 304 [OTA] Creating MQTT Client...
6 304 [OTA] Connecting to broker...
I (4353) wifi: pm start, type:0

I (8173) PKCS11: Initializing SPIFFS
I (8183) PKCS11: Partition size: total: 52961, used: 0
7 1277 [OTA] Connected to broker.
8 1280 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/
<Your_Thing_Name>/jobs/$next/get/accepted
I (12963) ota_pal: prvPAL_GetPlatformImageState
I (12963) esp_ota_ops: [0] aflags/seq:0x2/0x1, pflags/seq:0xffffffff/0x0
```

```
9 1285 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/
<Your_Thing_Name>/jobs/notify-next
10 1286 [OTA Task] [OTA_CheckForUpdate] Request #0
11 1289 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken:
 0:<Your_Thing_Name> ]
12 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
13 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
14 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
15 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: afr_ota
16 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
17 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: files
18 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
19 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
20 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
21 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
22 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha256-ecdsa
23 1289 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
24 1289 [OTA Task] [prvOTA_Close] Context->0x3ffbb4a8
25 1290 [OTA] [OTA_AgentInit] Ready.
26 1390 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
27 1490 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
28 1590 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
29 1690 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
[ ... ]
```

4. The ESP32 board is now listening for OTA updates. The ESP-IDF monitor is launched by the `make flash monitor` command. You can press **Ctrl+]** to quit. You can also use your favorite TTY terminal program (for example, PuTTY, Tera Term, or GNU Screen) to listen to the board's serial output. Be aware that connecting to the board's serial port might cause it to reboot.

Install the initial version of firmware on the Nordic nRF52840 DK

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This guide is written with the assumption that you have already performed the steps in Getting started with the Nordic nRF52840-DK and Over-the-Air Update Prerequisites. Before you attempt an OTA update, you might want to run the MQTT demo project described in Getting Started with FreeRTOS to ensure that your board and toolchain are set up correctly.

To flash an initial factory image to the board

1. Open *freertos*/vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/aws_demo_config.h.
2. Replace `#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED` with `CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED` or `CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED`.
3. With the OTA Update demo selected, follow the same steps outlined in Getting started with the Nordic nRF52840-DK to build and flash the image.

   You should see output similar to the following.

```
9 1285 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/your-thing-name/jobs/notify-next
10 1286 [OTA Task] [OTA_CheckForUpdate] Request #0
11 1289 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken: 0:your-thing-name ]
12 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
13 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
14 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
15 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: afr_ota
16 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
17 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: files
18 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
19 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
20 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
21 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
22 1289 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha256-ecdsa
23 1289 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
24 1289 [OTA Task] [prvOTA_Close] Context->0x3ffbb4a8
25 1290 [OTA] [OTA_AgentInit] Ready.
26 1390 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
27 1490 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
28 1590 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
29 1690 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
```

Your board is now listening for OTA updates.

Initial firmware on the Windows simulator

When you use the Windows simulator, there is no need to flash an initial version of the firmware. The Windows simulator is part of the `aws_demos` application, which also includes the firmware.

Install the initial version of firmware on a custom board

Using your IDE, build the `aws_demos` project, making sure to include the OTA library. For more information about the structure of the FreeRTOS source code, see FreeRTOS demos.

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

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

Update the version of your firmware

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

1. Open the `aws_demos` project in your IDE.
2. Locate the file `/vendors/vendor/boards/board/aws_demos/config_files/ota_demo_config.h` and increment the value of **APP_VERSION_BUILD**.
3. To schedule an update to a Renesas rx65n platform with a filetype other than 0 (non-firmware files), you must sign the file with the Renesas Secure Flash Programmer tool, otherwise it will fail the signature check on the device. The tool creates a signed file package with the extension `.rsu` which is a proprietary file type for Renesas. The tool can be found on Github. You can use the following example command to generate the image:

   ```
   "Renesas Secure Flash Programmer.exe" CUI Update "RX65N(ROM 2MB)/Secure
    Bootloader=256KB" "sig-sha256-ecdsa" 1 "file_name" "output_file_name.rsu"
   ```

4. Rebuild the project.

You must copy your firmware update into the Amazon S3 bucket that you created as described in Create an Amazon S3 bucket to store your update. The name of the file you need to copy to Amazon S3 depends on the hardware platform you are using:

- Texas Instruments CC3220SF-LAUNCHXL: `vendors/ti/boards/cc3220_launchpad/aws_demos/ccs/debug/aws_demos.bin`
- Espressif ESP32: `vendors/espressif/boards/esp32/aws_demos/make/build/aws_demos.bin`

Creating an OTA update (AWS IoT console)

1. In the navigation pane of the AWS IoT console, under **Manage** select **Remote actions**, and then choose **Jobs**.
2. Choose **Create job**.
3. Under **Job type** select **Create FreeRTOS OTA update job**, then choose **Next**.
4. In **Job properties**, enter a **Job name** and (optionally) enter a **Description** of the job, then choose **Next**.
5. You can deploy an OTA update to a single device or a group of devices. Under **Devices to update**, choose one or more things or thing groups from the dropdown.
6. Under **Select the protocol for the file transfer**, select either **HTTP** or **MQTT**, or select both to allow each device to determine the protocol to use.
7. Under **Sign and choose your file**, select **Sign a new file for me**.
8. Under **Code signing profile**, choose **Create new profile**.
9. In **Create a code signing profile**, enter a name for your code-signing profile.

    a. Under **Device hardware platform**, choose your hardware platform.

> **Note**
>
> Only hardware platforms that have been qualified for FreeRTOS are displayed in this list. If you are testing a non-qualified platform, and you are using the ECDSA P-256 SHA-256 ciphersuite for signing, you can pick the Windows Simulator code signing profile to produce a compatible signature. If you are using a non-qualified platform, and you are using a ciphersuite other than ECDSA P-256 SHA-256 for

signing, you can use Code Signing for AWS IoT, or you can sign your firmware update yourself. For more information, see Digitally signing your firmware update.

b. Under **Code signing certificate**, choose **Select an existing certificate** and then select a previously imported certificate, or choose **Import a new code signing certificate**, choose your files and select **Import** to import a new certificate.

c. Under **Pathname of code signing certificate on device**, enter the fully qualified path name to the code signing certificate on your device. For most devices you can leave this field blank. For the Windows simulator and for devices that do place the certificate in a specific file location, enter the pathname here.

> **Important**
>
> On the Texas Instruments CC3220SF-LAUNCHXL, do not include a leading forward slash (**/**) in front of the file name if your code signing certificate exists in the root of the file system. Otherwise, the OTA update fails during authentication with a `file not found` error.

d. Select **Create**.

10. Under **File** select **Select an existing file** then choose **Browse S3**. A list of your Amazon S3 buckets is displayed. Choose the bucket that contains your firmware update, and then choose your firmware update in the bucket.

> **Note**
>
> The Microchip Curiosity PIC32MZEF demo projects produce two binary images with default names of `mplab.production.bin` and `mplab.production.ota.bin`. Use the second file when you upload an image for OTA updating.

11. Under **Pathname of file on device**, enter the fully qualified path name to the location on your device where the OTA job will copy the firmware image. This location varies by platform.

> **Important**
>
> On the Texas Instruments CC3220SF-LAUNCHXL, due to security restrictions, the firmware image path name must be `/sys/mcuflashimg.bin`.

12. Open **File Type** and enter an integer value in the range 0-255. The file type you enter will be added to the Job document that is delivered to the MCU. The MCU firmware/software developer has full ownership on what to do with this value. Possible scenarios include an MCU that has a secondary processor whose firmware can be updated independently from the primary processor. When the device receives an OTA update job, it can use the File Type to identify which processor the update is for.
13. Under **IAM role**, choose a role according to the instructions in Create an OTA Update service role.
14. Choose **Next**.
15. Enter an ID and description for your OTA update job.
16. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.
17. Choose any appropriate optional configurations for your job (**Job executions rollout**, **Job abort**, **Job executions timeout**, and **Tags**).
18. Choose **Create**.

To use a previously signed firmware image

1. Under **Select and sign your firmware image**, choose **Select a previously signed firmware image**.
2. Under **Pathname of firmware image on device**, enter the fully qualified path name to the location on your device where the OTA job will copy the firmware image. This location varies by platform.
3. Under **Previous code signing job**, choose **Select**, and then choose the previous code-signing job used to sign the firmware image you are using for the OTA update.

Using a custom signed firmware image

1. Under **Select and sign your firmware image**, choose **Use my custom signed firmware image**.
2. Under **Pathname of code signing certificate on device**, enter the fully qualified path name to the code signing certificate on your device. For most devices you can leave this field blank. For the Windows simulator and for devices that do place the certificate in a specific file location, enter the pathname here.

3. Under **Pathname of firmware image on device**, enter the fully qualified path name to the location on your device where the OTA job will copy the firmware image. This location varies by platform.
4. Under **Signature**, paste your PEM format signature.
5. Under **Original hash algorithm**, choose the hash algorithm that was used when you created your file signature.
6. Under **Original encryption algorithm**, choose the algorithm that was used when you created your file signature.
7. Under **Select your firmware image in Amazon S3**, choose the Amazon S3 bucket and the signed firmware image in the Amazon S3 bucket.

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

> **Note**
>
> Do not use any personally identifiable information in the job ID for your OTA update. Examples of personally identifiable information include:
>
> - Names.
> - IP addresses.
> - Email addresses.
> - Locations.
> - Bank details.
> - Medical information.

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

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

Note

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

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

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

For a tutorial that shows you how to use the console to create an OTA update, see Over-the-air updates demo application.

Creating an OTA update with the AWS CLI

When you use the AWS CLI to create an OTA update, you:

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

Digitally signing your firmware update

When you use the AWS CLI to perform OTA updates, you can use Code Signing for AWS IoT, or you can sign your firmware update yourself. For a list of the cryptographic signing and hashing algorithms supported by Code Signing for AWS IoT, see SigningConfigurationOverrides. If you want to use a cryptographic algorithm that is not supported by Code Signing for AWS IoT, you must sign your firmware binary before you upload it to Amazon S3.

Signing your firmware image with Code Signing for AWS IoT

To sign your firmware image using Code Signing for AWS IoT, you can use one of the AWS SDKs or command line tools. For more information about Code Signing for AWS IoT, see Code Signing for AWS IoT.

After you install and configure the code-signing tools, copy your unsigned firmware image to your Amazon S3 bucket and start a code-signing job with the following AWS CLI commands. The **put-**

signing-profile command creates a reusable code-signing profile. The **start-signing-job** command starts the signing job.

```
aws signer put-signing-profile \
    --profile-name your_profile_name \
    --signing-material certificateArn=arn:aws:acm::your-region:your-aws-account-
id:certificate/your-certificate-id \
    --platform your-hardware-platform \
    --signing-parameters certname=your_certificate_path_on_device
```

```
aws signer start-signing-job \
    --source
 's3={bucketName=your_s3_bucket,key=your_s3_object_key,version=your_s3_object_version_id}'
 \
    --destination 's3={bucketName=your_destination_bucket}' \
    --profile-name your_profile_name
```

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

These are the parameters for the **put-signing-profile** and **start-signing-job** commands:

`source`

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

- `bucketName`: The name of your S3 bucket.
- `key`: The key (file name) of your firmware in your S3 bucket.
- `version`: The S3 version of your firmware in your S3 bucket. This is different from your firmware version. You can find it by browsing to the Amazon S3 console, choosing your bucket, and at the top of the page, next to **Versions**, choosing **Show**.

`destination`

The destination on the device to which the signed firmware in the S3 bucket will be copied. The format of this parameter is the same as the `source` parameter.

`signing-material`

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

`signing-parameters`

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

> **Note**
>
> This parameter is required when you are creating a code-signing profile for signing OTA updates with Code Signing for AWS IoT.

`platform`

The `platformId` of the hardware platform to which you are distributing the OTA update.

To return a list of the available platforms and their `platformId` values, use the `aws signer list-signing-platforms` command.

The signing job starts and writes the signed firmware image into the destination Amazon S3 bucket. The file name for the signed firmware image is a GUID. You need this file name when you create a stream. You can find the file name by browsing to the Amazon S3 console and choosing your bucket. If you don't see a file with a GUID file name, refresh your browser.

The command displays a job ARN and job ID. You need these values later on. For more information about Code Signing for AWS IoT, see Code Signing for AWS IoT.

Signing your firmware image manually

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

Creating a stream of your firmware update

A stream is an abstract interface to data that can be consumed by a device. A stream can hide the complexity of accessing data stored in different locations or different cloud-based services. The

OTA Update Manager service enables you to use multiple pieces of data, stored in various locations in Amazon S3, to perform an OTA Update.

When you create an AWS IoT OTA Update, you can also create a stream that contains your signed firmware update. Make a JSON file (`stream.json`) that identifies your signed firmware image. The JSON file should contain the following.

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

These are the attributes in the JSON file:

`fileId`

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

`s3Location`

The bucket and key for the firmware to stream.

`bucket`

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

`key`

The file name of your signed firmware image in the Amazon S3 bucket. You can find this value in the Amazon S3 console by looking at the contents of your bucket.

If you are using Code Signing for AWS IoT, the file name is a GUID generated by Code Signing for AWS IoT.

Use the **create-stream** AWS CLI command to create a stream.

```
aws iot create-stream \
    --stream-id your_stream_id \
```

```
    --description your_description \
    --files file://stream.json \
    --role-arn your_role_arn
```

These are the arguments for the **create-stream** AWS CLI command:

`stream-id`

An arbitrary string to identify the stream.

`description`

An optional description of the stream.

`files`

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

`fileId`

An arbitrary file ID.

`s3Location`

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

`bucket`

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

`key`

The key (file name) of the signed firmware image.

When you use Code Signing for AWS IoT, this key is a GUID.

The following is an example `stream.json` file.

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

```
        }
    }
]
```

`role-arn`

The OTA service role that also grants access to the Amazon S3 bucket where the firmware image is stored.

To find the Amazon S3 object key of your signed firmware image, use the **aws signer describe-signing-job --job-id *my-job-id*** command where `my-job-id` is the job ID displayed by the **create-signing-job** AWS CLI command. The output of the **describe-signing-job** command contains the key of the signed firmware image.

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

Creating an OTA update

Use the **create-ota-update** AWS CLI command to create an OTA update job.

> **Note**
>
> Do not use any personally identifiable information (PII) in your OTA update job ID. Examples of personally identifiable information include:
>
> - Names.
> - IP addresses.
> - Email addresses.
> - Locations.
> - Bank details.
> - Medical information.

```
aws iot  create-ota-update \
    --ota-update-id value \
    [--description value] \
    --targets value \
    [--protocols value] \
    [--target-selection value] \
    [--aws-job-executions-rollout-config value] \
    [--aws-job-presigned-url-config value] \
    [--aws-job-abort-config value] \
    [--aws-job-timeout-config value] \
    --files value \
    --role-arn value \
    [--additional-parameters value] \
    [--tags value]  \
    [--cli-input-json value] \
    [--generate-cli-skeleton]
```

`cli-input-json` format

```
{
  "otaUpdateId": "string",
  "description": "string",
  "targets": [
    "string"
  ],
  "protocols": [
    "string"
  ],
  "targetSelection": "string",
  "awsJobExecutionsRolloutConfig": {
    "maximumPerMinute": "integer",
    "exponentialRate": {
      "baseRatePerMinute": "integer",
      "incrementFactor": "double",
      "rateIncreaseCriteria": {
        "numberOfNotifiedThings": "integer",
        "numberOfSucceededThings": "integer"
      }
    }
  },
  "awsJobPresignedUrlConfig": {
    "expiresInSec": "long"
  },
```

```
  "awsJobAbortConfig": {
    "abortCriteriaList": [
      {
        "failureType": "string",
        "action": "string",
        "thresholdPercentage": "double",
        "minNumberOfExecutedThings": "integer"
      }
    ]
  },
  "awsJobTimeoutConfig": {
    "inProgressTimeoutInMinutes": "long"
  },
  "files": [
    {
      "fileName": "string",
      "fileType": "integer",
      "fileVersion": "string",
      "fileLocation": {
        "stream": {
          "streamId": "string",
          "fileId": "integer"
        },
        "s3Location": {
          "bucket": "string",
          "key": "string",
          "version": "string"
        }
      },
      "codeSigning": {
        "awsSignerJobId": "string",
        "startSigningJobParameter": {
          "signingProfileParameter": {
            "certificateArn": "string",
            "platform": "string",
            "certificatePathOnDevice": "string"
          },
          "signingProfileName": "string",
          "destination": {
            "s3Destination": {
              "bucket": "string",
              "prefix": "string"
            }
          }
```

```
        },
        "customCodeSigning": {
          "signature": {
            "inlineDocument": "blob"
          },
          "certificateChain": {
            "certificateName": "string",
            "inlineDocument": "string"
          },
          "hashAlgorithm": "string",
          "signatureAlgorithm": "string"
        }
      },
      "attributes": {
        "string": "string"
      }
    }
  ],
  "roleArn": "string",
  "additionalParameters": {
    "string": "string"
  },
  "tags": [
    {
      "Key": "string",
      "Value": "string"
    }
  ]
}
```

`cli-input-json` fields

Name	Type	Description
`otaUpdateId`	string (max:128 min:1)	The ID of the OTA update to be created.
`description`	string (max:2028)	The description of the OTA update.

Name	Type	Description
`targets`	list	The devices targeted to receive OTA updates.
`protocols`	list	The protocol used to transfer the OTA update image. Valid values are [HTTP], [MQTT], [HTTP, MQTT]. When both HTTP and MQTT are specified , the target device can choose the protocol.
`targetSelection`	string	Specifies whether the update will continue to run (CONTINUOUS), or will be complete after all the things specified as targets have completed the update (SNAPSHOT). If continuous, the update may also be run on a thing when a change is detected in a target. For example, an update will run on a thing when the thing is added to a target group, even after the update was completed by all things originally in the group. Valid values: CONTINUOUS | SNAPSHOT. enum: CONTINUOUS | SNAPSHOT
`awsJobExecutionsRolloutConfig`		Configuration for the rollout of OTA updates.

Name	Type	Description
maximumPerMinute	integer (max:1000 min:1)	The maximum number of OTA update job executions started per minute.
exponentialRate		The rate of increase for a job rollout. This parameter allows you to define an exponential rate increase for a job rollout.
baseRatePerMinute	integer (max:1000 min:1)	The minimum number of things that will be notified of a pending job, per minute, at the start of the job rollout. This is the initial rate of the rollout.
rateIncreaseCriteria		The criteria to initiate the increase in rate of rollout for a job. AWS IoT supports up to one digit after the decimal (for example, 1.5, but not 1.55).
numberOfNotifiedThings	integer (min:1)	When this number of things have been notified, it will initiate an increase in the rollout rate.
numberOfSucceededThings	integer (min:1)	When this number of things have succeeded in their job execution, it will initiate an increase in the rollout rate.
awsJobPresignedUrlConfig		Configuration information for pre-signed URLs.

Name	Type	Description
`expiresInSec`	long	How long (in seconds) pre-signed URLs are valid. Valid values are 60 - 3600, the default value is 1800 seconds. Pre-signed URLs are generated when a request for the job document is received.
`awsJobAbortConfig`		The criteria that determine when and how a job stoppage takes place.
`abortCriteriaList`	list	The list of criteria that determine when and how to stop the job.
`failureType`	string	The type of job execution failures that can initiate a job stoppage. enum: FAILED | REJECTED | TIMED_OUT | ALL
`action`	string	The type of job action to take to initiate the job stoppage. enum: CANCEL
`minNumberOfExecutedThings`	integer (min:1)	The minimum number of things which must receive job execution notifications before the job can be stopped.

Name	Type	Description
awsJobTimeoutConfig		Specifies the amount of time each device has to finish its execution of the job. A timer is started when the job execution status is set to IN_PROGRESS . If the job execution status is not set to another terminal state before the timer expires, it will be automatically set to TIMED_OUT .
inProgressTimeoutInMinutes	long	Specifies the amount of time, in minutes, this device has to finish execution of this job. The timeout interval can be anywhere between 1 minute and 7 days (1 to 10080 minutes). The in progress timer can't be updated and will apply to all job executions for the job. Whenever a job execution remains in the IN_PROGRESS status for longer than this interval, the job execution will fail and switch to the terminal TIMED_OUT status.
files	list	The files to be streamed by the OTA update.
fileName	string	The name of the file.

Name	Type	Description
`fileType`	integer range- max:255 min:0	An integer value you can include in the job document to allow your devices to identify the type of file received from the cloud.
`fileVersion`	string	The file version.
`fileLocation`		The location of the updated firmware.
`stream`		The stream that contains the OTA update.
`streamId`	string (max:128 min:1)	The stream ID.
`fileId`	integer (max:255 min:0)	The ID of a file associated with a stream.
`s3Location`		The location of the updated firmware in S3.
`bucket`	string (min:1)	The S3 bucket.
`key`	string (min:1)	The S3 key.
`version`	string	The S3 bucket version.
`codeSigning`		The code signing method of the file.

Name	Type	Description
awsSignerJobId	string	The ID of the AWSSignerJob which was created to sign the file.
startSigningJobParameter		Describes the code-signing job.
signingProfileParameter		Describes the code-signing profile.
certificateArn	string	Certificate ARN.
platform	string	The hardware platform of your device.
certificatePathOnDevice	string	The location of the code-signing certificate on your device.
signingProfileName	string	The code-signing profile name.
destination		The location to write the code-signed file.
s3Destination		Describes the location in S3 of the updated firmware.
bucket	string (min:1)	The S3 bucket that contains the updated firmware.
prefix	string	The S3 prefix.
customCodeSigning		A custom method for code signing a file.
signature		The signature for the file.

Name	Type	Description
inlineDocument	blob	A base64 encoded binary representation of the code signing signature.
certificateChain		The certificate chain.
certificateName	string	The name of the certificate.
inlineDocument	string	A base64 encoded binary representation of the code signing certificate chain.
hashAlgorithm	string	The hash algorithm used to code sign the file.
signatureAlgorithm	string	The signature algorithm used to code sign the file.
attributes	map	A list of name/attribute pairs.
roleArn	string (max:2048 min:20)	The IAM role that grants AWS IoT access to the Amazon S3, AWS IoT jobs and AWS Code Signing resources to create an OTA update job.
additionalParameters	map	A list of additional OTA update parameters which are name-value pairs.
tags	list	Metadata which can be used to manage updates.
Key	string (max:128 min:1)	The tag's key.

Name	Type	Description
Value	string (max:256 min:1)	The tag's value.

Output

```
{
  "otaUpdateId": "string",
  "awsIotJobId": "string",
  "otaUpdateArn": "string",
  "awsIotJobArn": "string",
  "otaUpdateStatus": "string"
}
```

AWS CLI output fields

Name	Type	Description
otaUpdateId	string (max:128 min:1)	The OTA update ID.
awsIotJobId	string	The AWS IoT job ID associated with the OTA update.
otaUpdateArn	string	The OTA update ARN.
awsIotJobArn	string	The AWS IoT job ARN associated with the OTA update.
otaUpdateStatus	string	The OTA update status. enum: CREATE_PENDING | CREATE_IN_PROGRESS | CREATE_COMPLETE | CREATE_FAILED

The following is an example of a JSON file passed into the **create-ota-update** command that uses Code Signing for AWS IoT.

```
[
  {
    "fileName": "firmware.bin",
    "fileType": 1,
    "fileLocation": {
      "stream": {
        "streamId": "004",
        "fileId":123
      }
    },
    "codeSigning": {
      "awsSignerJobId": "48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
    }
  }
]
```

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

```
[
  {
    "fileName": "firmware.bin",
    "fileType": 1,
    "fileLocation": {
      "stream": {
        "streamId": "004",
        "fileId": 123
      }
    },
    "codeSigning": {
      "customCodeSigning":{
        "signature":{
          "inlineDocument":"your_signature"
        },
        "certificateChain": {
          "certificateName": "your_certificate_name",
          "inlineDocument":"your_certificate_chain"
        },
        "hashAlgorithm":"your_hash_algorithm",
        "signatureAlgorithm":"your_signature_algorithm"
```

```
      }
    }
  }
]
```

The following is an example of a JSON file passed into the **create-ota-update** AWS CLI command that allows FreeRTOS OTA to start a code-signing job and create a code-signing profile and stream.

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "your_bucket_name",
        "key": "your_object_key",
        "version": "your_S3_object_version"
      }
    },
    "codeSigning":{
      "startSigningJobParameter":{
        "signingProfileName": "myTestProfile",
        "signingProfileParameter": {
          "certificateArn": "your_certificate_arn",
          "platform": "your_platform_id",
          "certificatePathOnDevice": "certificate_path"
        },
        "destination": {
          "s3Destination": {
            "bucket": "your_destination_bucket"
          }
        }
      }
    }
  }
]
```

The following is an example of a JSON file passed into the **create-ota-update** AWS CLI command that creates an OTA update that starts a code-signing job with an existing profile and uses the specified stream.

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "your_s3_bucket_name",
        "key": "your_object_key",
        "version": "your_S3_object_version"
      }
    },
    "codeSigning":{
      "startSigningJobParameter":{
        "signingProfileName": "your_unique_profile_name",
        "destination": {
          "s3Destination": {
            "bucket": "your_destination_bucket"
          }
        }
      }
    }
  }
]
```

The following is an example of a JSON file passed into the **create-ota-update** AWS CLI command that allows FreeRTOS OTA to create a stream with an existing code-signing job ID.

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "codeSigning":{
      "awsSignerJobId": "your_signer_job_id"
    }
  }
]
```

The following is an example of a JSON file passed into the **create-ota-update** AWS CLI command that creates an OTA update. The update creates a stream from the specified S3 object and uses custom code signing.

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "your_bucket_name",
        "key": "your_object_key",
        "version": "your_S3_object_version"
      }
    },
    "codeSigning":{
      "customCodeSigning": {
        "signature":{
          "inlineDocument":"your_signature"
        },
        "certificateChain": {
          "inlineDocument":"your_certificate_chain",
          "certificateName": "your_certificate_path_on_device"
        },
        "hashAlgorithm":"your_hash_algorithm",
        "signatureAlgorithm":"your_sig_algorithm"
      }
    }
  }
]
```

Listing OTA updates

You can use the **list-ota-updates** AWS CLI command to get a list of all OTA updates.

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

The output from the **list-ota-updates** command looks like this.

```
{
  "otaUpdates": [

    {
      "otaUpdateId": "my_ota_update2",
      "otaUpdateArn": "arn:aws:iot:us-west-2:123456789012:otaupdate/my_ota_update2",
```

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

Getting information about an OTA update

You can use the **get-ota-update** AWS CLI command to get the creation or deletion status of an OTA update.

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

The output from the **get-ota-update** command looks like the following.

```
{
    "otaUpdateInfo": {
        "otaUpdateId": "ota-update-001",
        "otaUpdateArn": "arn:aws:iot:region:123456789012:otaupdate/ota-update-001",
        "creationDate": 1575414146.286,
        "lastModifiedDate": 1575414149.091,
        "targets": [
            "arn:aws:iot:region:123456789012:thing/myDevice"
        ],
        "protocols": [ "HTTP" ],
        "awsJobExecutionsRolloutConfig": {
            "maximumPerMinute": 0
        },
        "awsJobPresignedUrlConfig": {
            "expiresInSec": 1800
        },
        "targetSelection": "SNAPSHOT",
        "otaUpdateFiles": [
```

```
            {
                "fileName": "my_firmware.bin",
                "fileType": 1,
                "fileLocation": {
                    "s3Location": {
                        "bucket": "my-bucket",
                        "key": "my_firmware.bin",
                        "version": "AvP3bfJC9gyqnwoxPHuTqM5GWENt4iii"
                    }
                },
                "codeSigning": {
                    "awsSignerJobId": "b7a55a54-fae5-4d3a-b589-97ed103737c2",
                    "startSigningJobParameter": {
                        "signingProfileParameter": {},
                        "signingProfileName": "my-profile-name",
                        "destination": {
                            "s3Destination": {
                                "bucket": "some-ota-bucket",
                                "prefix": "SignedImages/"
                            }
                        }
                    },
                    "customCodeSigning": {}
                }
            }
        ],
        "otaUpdateStatus": "CREATE_COMPLETE",
        "awsIotJobId": "AFR_OTA-ota-update-001",
        "awsIotJobArn": "arn:aws:iot:region:123456789012:job/AFR_OTA-ota-update-001"
    }
}
```

The values returned for `otaUpdateStatus` include the following:

`CREATE_PENDING`

The creation of an OTA update is pending.

`CREATE_IN_PROGRESS`

An OTA update is being created.

`CREATE_COMPLETE`

An OTA update has been created.

`CREATE_FAILED`

The creation of an OTA update failed.

`DELETE_IN_PROGRESS`

An OTA update is being deleted.

`DELETE_FAILED`

The deletion of an OTA update failed.

> **Note**
>
> To get the execution status of an OTA update after it is created, you need to use the **describe-job-execution** command. For more information, see Describe Job Execution.

Deleting OTA-related data

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

Deleting an OTA stream

When you create an OTA update that uses MQTT, either you can use the command-line or the AWS IoT console to create a stream to break the firmware up into chunks so it can be sent over MQTT. You can delete this stream with the **delete-stream** AWS CLI command, as shown in the following example.

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

Deleting an OTA update

When you create an OTA update, the following are created:

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

The **delete-ota-update** command deletes the entry in the OTA update job database only. You must use the **delete-job** command to delete the AWS IoT job.

Use the **delete-ota-update** command to delete an OTA update.

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

`ota-update-id`

The ID of the OTA update to delete.

`delete-stream`

Deletes the stream associated with the OTA update.

`force-delete-aws-job`

Deletes the AWS IoT job associated with the OTA update. If this flag is not set and the job is in the `In_Progress` state, the job is not deleted.

Deleting an IoT job created for an OTA update

FreeRTOS creates an AWS IoT job when you create an OTA update. A job execution is also created for each device that processes the job. You can use the **delete-job** AWS CLI command to delete a job and its associated job executions.

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

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

> **Note**
>
> Deleting a job with a status of IN_PROGRESS interrupts any job executions that are IN_PROGRESS on your devices and can result in a device being left in a nondeterministic state. Make sure that each device executing a job that has been deleted can recover to a known state.

Depending on the number of job executions created for the job and other factors, it can take a few minutes to delete a job. While the job is being deleted, its status is DELETION_IN_PROGRESS. Attempting to delete or cancel a job whose status is already DELETION_IN_PROGRESS results in an error.

You can use the **delete-job-execution** to delete a job execution. You might want to delete a job execution when a small number of devices are unable to process a job. This deletes the job execution for a single device, as shown in the following example.

```
aws iot delete-job-execution --job-id your-job-id --thing-name
                    your-thing-name --execution-number your-job-execution-number --no-
force
```

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

> **Note**
>
> Deleting a job execution with a status of IN_PROGRESS interrupts any job executions that are IN_PROGRESS on your devices and can result in a device being left in a nondeterministic state. Make sure that each device executing a job that has been deleted can recover to a known state.

For more information about using the OTA update demo application, see Over-the-air updates demo application.

OTA Update Manager service

The over-the-air (OTA) Update Manager service provides a way to:

- Create an OTA update and the resources it uses, including an AWS IoT job, an AWS IoT stream, and code signing.
- Get information about an OTA update.
- List all OTA updates associated with your AWS account.
- Delete an OTA update.

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

- An OTA update ID.
- An optional OTA update description.
- A list of devices to update (*targets*).
- The type of OTA update: CONTINUOUS or SNAPSHOT. See the Jobs section of the *AWS IoT Developer Guide* for a discussion of the type of update that you need.
- The protocol used to perform the OTA update: [MQTT], [HTTP] or [MQTT, HTTP]. When you specify MQTT and HTTP, the device setup determines the protocol used.
- A list of files to send to the target devices.
- The IAM role that grants AWS IoT access to the Amazon S3, AWS IoT jobs and AWS Code Signing resources to create an OTA update job.
- An optional list of user-defined name-value pairs.

OTA updates were designed to update device firmware, but you can use them to send any files that you want to one or more devices registered with AWS IoT. When you send firmware updates over the air, we recommend that you digitally sign them so that the devices that receive them can verify they haven't been tampered with en route.

You can send updated firmware images using the HTTP or MQTT protocol, depending on the settings that you choose. You can sign your firmware updates with Code Signing for FreeRTOS or you can use your own code-signing tools.

For more control over the process, you can use the CreateStream API to create a stream when sending updates over MQTT. In some instances, you can modify the FreeRTOS Agent code to adjust the size of the blocks that you send and receive.

When you create an OTA update, the OTA Manager service creates an AWS IoT job to notify your devices that an update is available. The FreeRTOS OTA Agent runs on your devices and listens for update messages. When an update is available, it requests the firmware update image over HTTP or MQTT and stores the files locally. It checks the digital signature of the downloaded files and, if valid, installs the firmware update. If you're not using FreeRTOS, you must implement your own OTA Agent to listen for and download updates and perform any installation operations.

Integrating the OTA Agent into your application

The over-the-air (OTA) Agent is designed to simplify the amount of code you must write to add OTA update functionality to your product. That integration burden consists primarily of initialization of the OTA Agent and creating a custom callback function for responding to the OTA Agent event messages. During initialization OS, MQTT, HTTP (if HTTP is used for file download) and platform specific implementation (PAL) interfaces are passed to the OTA Agent. Buffers can also be initialized and passed to the OTA Agent.

> **Note**
>
> Although the integration of the OTA update feature into your application is rather simple, the OTA update system requires an understanding of more than just device code integration. To familiarize yourself with how to configure your AWS account with AWS IoT things, credentials, code-signing certificates, provisioning devices, and OTA update jobs, see FreeRTOS Prerequisites.

Connection management

The OTA Agent uses the MQTT protocol for all control communication operations involving AWS IoT services, but it doesn't manage the MQTT connection. To ensure that the OTA Agent doesn't interfere with the connection management policy of your application, the MQTT connection (including disconnect and any reconnect functionality) must be handled by the main user application. The file can be downloaded over the MQTT or HTTP protocol. You can choose which protocol when you create the OTA job. If you choose MQTT, the OTA Agent uses the same connection for control operations and for downloading files.

Simple OTA demo

The following is an excerpt of a simple OTA demo that shows you how the Agent connects to the MQTT broker and initializes the OTA Agent. In this example, we configure the demo to use the default OTA application callback and to return some statistics once per second. For brevity, we leave out some details from this demo.

The OTA demo also demonstrates MQTT connection management by monitoring the disconnect callback and reestablishing the connection. When a disconnect happens, the demo first suspends the OTA Agent operations and then attempts to reestablish the MQTT connection. The MQTT reconnection attempts are delayed by a time which is exponentially increased up to a maximum

value and a jitter is also added. If the connection is reestablished, the OTA Agent continues its operations.

For a working example that uses the AWS IoT MQTT broker, see the OTA demo code in the `demos/ota` directory.

Because the OTA Agent is its own task, the intentional one-second delay in this example affects this application only. It has no impact on the performance of the Agent.

```
static BaseType_t prvRunOTADemo( void )
{
    /* Status indicating a successful demo or not. */
    BaseType_t xStatus = pdFAIL;

    /* OTA library return status. */
    OtaErr_t xOtaError = OtaErrUninitialized;

    /* OTA event message used for sending event to OTA Agent.*/
    OtaEventMsg_t xEventMsg = { 0 };

    /* OTA interface context required for library interface functions.*/
    OtaInterfaces_t xOtaInterfaces;

    /* OTA library packet statistics per job.*/
    OtaAgentStatistics_t xOtaStatistics = { 0 };

    /* OTA Agent state returned from calling OTA_GetState.*/
    OtaState_t xOtaState = OtaAgentStateStopped;

    /* Set OTA Library interfaces.*/
    prvSetOtaInterfaces( &xOtaInterfaces );

    /************************** Init OTA Library. **************************/

    if( ( xOtaError = OTA_Init( &xOtaBuffer,
                                &xOtaInterfaces,
                                ( const uint8_t * ) ( democonfigCLIENT_IDENTIFIER ),
                                prvOtaAppCallback ) ) != OtaErrNone )
    {
        LogError( ( "Failed to initialize OTA Agent, exiting = %u.",
                    xOtaError ) );
    }
    else
```

```
    {
        xStatus = pdPASS;
    }

    /************************ Create OTA Agent Task. ************************/

    if( xStatus == pdPASS )
    {
        xStatus = xTaskCreate( prvOTAAgentTask,
                               "OTA Agent Task",
                               otaexampleAGENT_TASK_STACK_SIZE,
                               NULL,
                               otaexampleAGENT_TASK_PRIORITY,
                               NULL );

        if( xStatus != pdPASS )
        {
            LogError( ( "Failed to create OTA agent task:" ) );
        }
    }

    /***************************** Start OTA *****************************/

    if( xStatus == pdPASS )
    {
        /* Send start event to OTA Agent.*/
        xEventMsg.eventId = OtaAgentEventStart;
        OTA_SignalEvent( &xEventMsg );
    }

    /******************** Loop and display OTA statistics ********************/

    if( xStatus == pdPASS )
    {
        while( ( xOtaState = OTA_GetState() ) != OtaAgentStateStopped )
        {
            /* Get OTA statistics for currently executing job. */
            if( xOtaState != OtaAgentStateSuspended )
            {
                OTA_GetStatistics( &xOtaStatistics );

                LogInfo( ( " Received: %u   Queued: %u   Processed: %u   Dropped: %u",
                           xOtaStatistics.otaPacketsReceived,
                           xOtaStatistics.otaPacketsQueued,
```

```
                            xOtaStatistics.otaPacketsProcessed,
                            xOtaStatistics.otaPacketsDropped ) );
            }

            vTaskDelay( pdMS_TO_TICKS( otaexampleEXAMPLE_TASK_DELAY_MS ) );
        }
    }

    return xStatus;
}
```

Here is the high-level flow of this demo application:

- Create an MQTT Agent context.
- Connect to your AWS IoT endpoint.
- Initialize the OTA Agent.
- Loop that allows an OTA update job and outputs statistics once a second.
- If MQTT disconnects, suspend the OTA Agent operations.
- Try connecting again with exponential delay and jitter.
- If reconnected, resume OTA Agent operations.
- If the Agent stops, delay one second, and then try reconnecting.

Using application callback for OTA Agent events

The previous example used `prvOtaAppCallback` as the callback handler for OTA Agent events. (See the fourth parameter to the `OTA_Init` API call). If you want to implement custom handling of the completion events, you must change the default handling in the OTA demo/application. During the OTA process, the OTA Agent can send one of the following event enums to the callback handler. It is up to the application developer to decide how and when to handle these events.

```
/**
 * @ingroup ota_enum_types
 * @brief OTA Job callback events.
 *
 * After an OTA update image is received and authenticated, the agent calls the user
 * callback (set with the @ref OTA_Init API) with the value OtaJobEventActivate to
 * signal that the device must be rebooted to activate the new image. When the device
 * boots, if the OTA job status is in self test mode, the agent calls the user callback
```

```
 * with the value OtaJobEventStartTest, signaling that any additional self tests
 * should be performed.
 *
 * If the OTA receive fails for any reason, the agent calls the user callback with
 * the value OtaJobEventFail instead to allow the user to log the failure and take
 * any action deemed appropriate by the user code.
 *
 * See the OtaImageState_t type for more information.
 */
typedef enum OtaJobEvent
{
    OtaJobEventActivate = 0,       /*!< @brief OTA receive is authenticated and ready
 to activate. */
    OtaJobEventFail = 1,           /*!< @brief OTA receive failed. Unable to use this
 update. */
    OtaJobEventStartTest = 2,      /*!< @brief OTA job is now in self test, perform
 user tests. */
    OtaJobEventProcessed = 3,      /*!< @brief OTA event queued by OTA_SignalEvent is
 processed. */
    OtaJobEventSelfTestFailed = 4, /*!< @brief OTA self-test failed for current job. */
    OtaJobEventParseCustomJob = 5, /*!< @brief OTA event for parsing custom job
 document. */
    OtaJobEventReceivedJob = 6,    /*!< @brief OTA event when a new valid AFT-OTA job
 is received. */
    OtaJobEventUpdateComplete = 7, /*!< @brief OTA event when the update is completed.
 */
    OtaLastJobEvent = OtaJobEventStartTest
} OtaJobEvent_t;
```

The OTA Agent can receive an update in the background during active processing of the main application. The purpose of delivering these events is to allow the application to decide if action can be taken immediately or if it should be deferred until after completion of some other application-specific processing. This prevents an unanticipated interruption of your device during active processing (for example, vacuuming) that would be caused by a reset after a firmware update. These are the job events received by the callback handler:

`OtaJobEventActivate`

When the callback handler receives this event, you can either reset the device immediately or schedule a call to reset the device later. This allows you to postpone the device reset and self-test phase, if necessary.

`OtaJobEventFail`

When the callback handler receives this event, the update has failed. You do not need to do anything in this case. You might want to output a log message or do something application-specific.

`OtaJobEventStartTest`

The self-test phase is meant to allow newly updated firmware to execute and test itself before it determines whether it is functioning properly and commit itself to be the latest permanent application image. When a new update is received and authenticated and the device has been reset, the OTA Agent sends the `OtaJobEventStartTest` event to the callback function when it is ready for testing. The developer can add any required tests to determine if the device firmware is functioning properly after update. When the device firmware is deemed reliable by the self tests, the code must commit the firmware as the new permanent image by calling the `OTA_SetImageState( OtaImageStateAccepted )` function.

`OtaJobEventProcessed`

The OTA event queued by `OTA_SignalEvent` is processed, so cleanup operations like freeing the OTA buffers can be done.

`OtaJobEventSelfTestFailed`

The OTA self-test failed for the current job. The default handling for this event is to shut down the OTA Agent and restart it so that the device rolls back to the previous image.

`OtaJobEventUpdateComplete`

The notification event for OTA job update completion.

OTA security

The following are three aspects of over-the-air (OTA) security:

Connection security

The OTA Update Manager service relies on existing security mechanisms, such as Transport Layer Security (TLS) mutual authentication, used by AWS IoT. OTA update traffic passes through the AWS IoT device gateway and uses AWS IoT security mechanisms. Each incoming and outgoing HTTP or MQTT message through the device gateway undergoes strict authentication and authorization.

Authenticity and integrity of OTA updates

Firmware can be digitally signed before an OTA update to ensure that it is from a reliable source and has not been tampered with.

The FreeRTOS OTA Update Manager service uses Code Signing for AWS IoT to automatically sign your firmware. For more information, see Code Signing for AWS IoT.

The OTA Agent, which runs on your devices, performs integrity checks on the firmware when it arrives on the device.

Operator security

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

Code Signing for AWS IoT

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

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

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

> **Note**
>
> You can download the Code Signing for AWS IoT SDK from Tools for Amazon Web Services.

OTA troubleshooting

The following sections contain information to help you troubleshoot issues with OTA updates.

Topics

- Set up CloudWatch Logs for OTA updates
- Log AWS IoT OTA API calls with AWS CloudTrail
- Get CreateOTAUpdate failure details using the AWS CLI
- Get OTA failure codes with the AWS CLI
- Troubleshoot OTA updates of multiple devices
- Troubleshoot OTA updates with the Texas Instruments CC3220SF Launchpad

Set up CloudWatch Logs for OTA updates

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

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

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

Create a logging role and enable logging

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

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

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

OTA update logs

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

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

You can view your logs in the CloudWatch console.

To view an OTA update in CloudWatch Logs

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

OTA update logs can contain the following properties:

accountId

The AWS account ID in which the log was generated.

actionType

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

- `CreateOTAUpdate`: An OTA update was created.
- `DeleteOTAUpdate`: An OTA update was deleted.
- `StartCodeSigning`: A code-signing job was started.
- `CreateAWSJob`: An AWS IoT job was created.

- `CreateStream`: A stream was created.
- `GetStream`: A request for a stream was sent to the AWS IoT MQTT-based file delivery feature.
- `DescribeStream`: A request for information about a stream was sent to the AWS IoT MQTT-based file delivery feature.

awsJobId

The AWS IoT job ID that generated the log.

clientId

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

clientToken

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

details

More information about the operation that generated the log.

logLevel

The logging level of the log. For OTA update logs, this is always set to `DEBUG`.

otaUpdateId

The ID of the OTA update that generated the log.

protocol

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

status

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

- Success
- Failure

streamId

The AWS IoT stream ID that generated the log.

timestamp

The time when the log was generated.

topicName

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

Example logs

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

```
{
    "timestamp": "2018-07-23 22:59:44.955",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "actionType": "StartCodeSigning",
    "otaUpdateId": "08957b03-eea3-448a-87fe-743e6891ca3a",
    "details": "Start code signing job. The request status is SUCCESS."
}
```

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

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

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

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

```
}
```

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

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

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

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

The following is an example log generated when a device requests a stream from the MQTT-based file delivery feature:

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

```
}
```

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

```
{
    "timestamp": "2018-07-25 22:10:12.690",
    "logLevel": "DEBUG",
    "accountId": "123456789012",
    "status": "Success",
    "actionType": "DescribeStream",
    "protocol": "MQTT",
    "clientId": "581075e0-4639-48ee-8b94-2cf304168e43",
    "topicName": "$aws/things/581075e0-4639-48ee-8b94-2cf304168e43/streams/71c101a8-
bcc5-4929-9fe2-af563af0c139/describe/json",
    "streamId": "71c101a8-bcc5-4929-9fe2-af563af0c139",
    "clientToken": "clientToken",
    "details": "The request status is SUCCESS."
}
```

Log AWS IoT OTA API calls with AWS CloudTrail

FreeRTOS is integrated with CloudTrail, a service that captures AWS IoT OTA API calls and delivers the log files to an Amazon S3 bucket that you specify. CloudTrail captures API calls from your code to the AWS IoT OTA APIs. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT OTA, the source IP address from which the request was made, who made the request, when it was made, and so on.

For more information about CloudTrail, including how to configure and enable it, see the *AWS CloudTrail User Guide*.

FreeRTOS information in CloudTrail

When CloudTrail logging is enabled in your AWS account, API calls made to AWS IoT OTA actions are tracked in CloudTrail log files where they are written with other AWS service records. CloudTrail determines when to create and write to a new file based on a time period and file size.

The following AWS IoT OTA control plane actions are logged by CloudTrail:

- CreateStream
- DescribeStream
- ListStreams

- UpdateStream
- DeleteStream
- CreateOTAUpdate
- GetOTAUpdate
- ListOTAUpdates
- DeleteOTAUpdate

> **Note**
>
> AWS IoT OTA data plane actions (device side) are not logged by CloudTrail. Use CloudWatch to monitor these.

Every log entry contains information about who generated the request. The user identity information in the log entry helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element. AWS IoT OTA actions are documented in the AWS IoT OTA API Reference.

You can store your log files in your Amazon S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

If you want to be notified when log files are delivered, you can configure CloudTrail to publish Amazon SNS notifications. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate AWS IoT OTA log files from multiple AWS Regions and multiple AWS accounts into a single Amazon S3 bucket.

For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

Understanding FreeRTOS log file entries

CloudTrail log files can contain one or more log entries. Each entry lists multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. Log entries are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the log from a call to `CreateOTAUpdate` action.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "EXAMPLE",
        "arn": "arn:aws:iam::your_aws_account:user/your_user_id",
        "accountId": "your_aws_account",
        "accessKeyId": "your_access_key_id",
        "userName": "your_username",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2018-08-23T17:27:08Z"
            }
        },
        "invokedBy": "apigateway.amazonaws.com"
    },
    "eventTime": "2018-08-23T17:27:19Z",
    "eventSource": "iot.amazonaws.com",
    "eventName": "CreateOTAUpdate",
    "awsRegion": "your_aws_region",
    "sourceIPAddress": "apigateway.amazonaws.com",
    "userAgent": "apigateway.amazonaws.com",
    "requestParameters": {
        "targets": [
            "arn:aws:iot:your_aws_region:your_aws_account:thing/Thing_CMH"
        ],
        "roleArn": "arn:aws:iam::your_aws_account:role/Role_FreeRTOSJob",
        "files": [
            {
                "fileName": "/sys/mcuflashimg.bin",
                "fileSource": {
                    "fileId": 0,
```

```
                    "streamId": "your_stream_id"
                },
                "codeSigning": {
                    "awsSignerJobId": "your_signer_job_id"
                }
            }
        ],
        "targetSelection": "SNAPSHOT",
        "otaUpdateId": "FreeRTOSJob_CMH-23-1535045232806-92"
    },
    "responseElements": {
        "otaUpdateArn": "arn:aws:iot:your_aws_region:your_aws_account:otaupdate/
FreeRTOSJob_CMH-23-1535045232806-92",
        "otaUpdateStatus": "CREATE_PENDING",
        "otaUpdateId": "FreeRTOSJob_CMH-23-1535045232806-92"
    },
    "requestID": "c9649630-a6f9-11e8-8f9c-e1cf2d0c9d8e",
    "eventID": "ce9bf4d9-5770-4cee-acf4-0e5649b845c0",
    "eventType": "AwsApiCall",
    "recipientAccountId": "recipient_aws_account"
}
```

Get CreateOTAUpdate failure details using the AWS CLI

If the process of creating an OTA update job fails, there may be actions you can take to remedy the problem. When you create an OTA update job, the OTA manager service creates an IoT job and schedules it for the target devices, and this process also creates or uses other types of AWS resources in your account (a code-signing job, an AWS IoT stream, an Amazon S3 object). Any error encountered may cause the process to fail without creating an AWS IoT job. In this troubleshooting section we give instructions on how to retrieve the details of the failure.

1. Install and configure the AWS CLI.
2. Run `aws configure` and enter the following information.

   ```
   $ aws configure
   AWS Access Key ID [None]: AccessID
   AWS Secret Access Key [None]: AccessKey
   Default region name [None]: Region
   Default output format [None]: json
   ```

   For more information, see Quick configuration with `aws configure`.

3. Run:

```
aws iot get-ota-update --ota-update-id ota_update_job_001
```

   Where *ota_update_job_001* is the ID you gave the OTA update when you created it.

4. The output will look like this:

```
{
    "otaUpdateInfo": {
        "otaUpdateId": "ota_update_job_001",
        "otaUpdateArn":
 "arn:aws:iot:region:account_id:otaupdate/ota_update_job_001",
        "creationDate": 1584646864.534,
        "lastModifiedDate": 1584646865.913,
        "targets": [
            "arn:aws:iot:region:account_id:thing/thing_001"
        ],
        "protocols": [
            "MQTT"
        ],
        "awsJobExecutionsRolloutConfig": {},
        "awsJobPresignedUrlConfig": {},
        "targetSelection": "SNAPSHOT",
        "otaUpdateFiles": [
            {
               "fileName": "/12ds",
                "fileLocation": {
                    "s3Location": {
                        "bucket": "bucket_name",
                        "key": "demo.bin",
                        "version": "Z7X.TWSAS7JSi4rybc02nMdcE41W1tV3"
                    }
                },
                "codeSigning": {
                    "startSigningJobParameter": {
                        "signingProfileParameter": {},
                        "signingProfileName": "signing_profile_name",
                        "destination": {
                            "s3Destination": {
                                "bucket": "bucket_name",
                                "prefix": "SignedImages/"
                            }
```

```
                        }
                    },
                    "customCodeSigning": {}
                }
            }
        ],
        "otaUpdateStatus": "CREATE_FAILED",
        "errorInfo": {
            "code": "AccessDeniedException",
            "message": "S3 object demo.bin not accessible. Please check
 your permissions (Service: AWSSigner; Status Code: 403; Error Code:
 AccessDeniedException; Request ID: 01d8e7a1-8c7c-4d85-9fd7-dcde975fdd2d)"
        }
    }
}
```

If the create failed, the `otaUpdateStatus` field in the command output will contain `CREATE_FAILED` and the `errorInfo` field will contain the details of the failure.

Get OTA failure codes with the AWS CLI

1. Install and configure the AWS CLI.
2. Run `aws configure` and enter following information.

```
$ aws configure
AWS Access Key ID [None]: AccessID
AWS Secret Access Key [None]: AccessKey
Default region name [None]: Region
Default output format [None]: json
```

For more information, see Quick configuration with `aws configure`.

3. Run:

```
aws iot describe-job-execution --job-id JobID --thing-name ThingName
```

Where *JobID* is the complete job ID string for the job whose status we want to get (it was associated with the OTA update job when it was created) and *ThingName* is the AWS IoT thing name that the device is registered as in AWS IoT

4. The output will look like this:

```
{
    "execution": {
        "jobId": "AFR_OTA-****************",
        "status": "FAILED",
        "statusDetails": {
            "detailsMap": {
                "reason": "0xEEEEEEEE: 0xffffffff"
            }
        },
        "thingArn": "arn:aws:iot:Region:AccountID:thing/ThingName",
        "queuedAt": 1569519049.9,
        "startedAt": 1569519052.226,
        "lastUpdatedAt": 1569519052.226,
        "executionNumber": 1,
        "versionNumber": 2
    }
}
```

In this example output, the "reason" in the "detailsmap" has two fields: the field shown as "0xEEEEEEEE" contains the generic error code from the OTA Agent; the field shown as "0xffffffff" contains the sub-code. The generic error codes are listed in https://docs.aws.amazon.com/freertos/latest/lib-ref/html1/aws__ota__agent_8h.html. See error codes with the prefix "kOTA_Err_". The sub-code can be a platform specific code or provide more details about the generic error.

Troubleshoot OTA updates of multiple devices

To perform OTAs on multiple devices (things) using the same firmware image, implement a function (for example `getThingName()`) that retrieves `clientcredentialIOT_THING_NAME` from non-volatile memory. Make sure that this function reads the thing name from a part of non-volatile memory that is not overwritten by the OTA, and that the thing name is provisioned before running the first job. If you are using the JITP flow, you can read the thing name out of your device certificate's common name.

Troubleshoot OTA updates with the Texas Instruments CC3220SF Launchpad

The CC3220SF Launchpad platform provides a software tamper-detection mechanism. It uses a security alert counter that is incremented whenever there is an integrity violation. The device is locked when the security alert counter reaches a predetermined threshold (the default is 15) and

the host receives the `SL_ERROR_DEVICE_LOCKED_SECURITY_ALERT` asynchronous event. The locked device then has limited accessibility. To recover the device, you can reprogram it or use the restore-to-factory process to revert to the factory image. You should program the desired behavior by updating the asynchronous event handler in `network_if.c`.

FreeRTOS Libraries

FreeRTOS libraries provide additional functionality to the FreeRTOS kernel and its internal libraries. You can use FreeRTOS libraries for networking and security in embedded applications. FreeRTOS libraries also enable your applications to interact with AWS IoT services. FreeRTOS includes libraries that make it possible to:

- Securely connect devices to the AWS IoT Cloud using MQTT and device shadows.
- Discover and connect to AWS IoT Greengrass cores.
- Manage Wi-Fi connections.
- Listen for and process FreeRTOS Over-the-Air Updates.

The `libraries` directory contains the source code of the FreeRTOS libraries. There are helper functions that assist in implementing the library functionality. We do not recommend that you change these helper functions.

FreeRTOS porting libraries

The following porting libraries are included in configurations of FreeRTOS that are available for download on the FreeRTOS console. These libraries are platform-dependent. Their contents change according to your hardware platform. For information about porting these libraries to a device, see the FreeRTOS Porting Guide.

FreeRTOS porting libraries

Library	API Reference	Description
Bluetooth Low Energy	Bluetooth Low Energy API Reference	Using the FreeRTOS Bluetooth Low Energy library, your microcontroller can communicate with the AWS IoT MQTT broker through a gateway device. For more information, see Bluetooth Low Energy library.

Library	API Reference	Description
Over-the-Air Updates	AWS IoT Over-the-air update API reference	The FreeRTOS AWS IoT Over-the-air (OTA) update library lets you manage update notificat ions, download updates, and perform cryptographic verificat ion of firmware updates on your FreeRTOS device. For more information, see AWS IoT Over the air (OTA) library.
FreeRTOS+POSIX	FreeRTOS+POSIX API Reference	You can use the FreeRTOS+POSIX library to port POSIX-compliant applications to the FreeRTOS ecosystem. For more information, see FreeRTOS+POSIX.
Secure Sockets	Secure Sockets API Reference	For more information, see Secure Sockets library.
FreeRTOS+TCP	FreeRTOS+TCP API Reference	FreeRTOS+TCP is a scalable, open source and thread safe TCP/IP stack for FreeRTOS. For more information, see FreeRTOS+TCP.
Wi-Fi	Wi-Fi API Reference	The FreeRTOS Wi-Fi library enables you to interface with your microcontroller's lower-level wireless stack. For more information, see the Wi-Fi library.

Library	API Reference	Description
corePKCS11		The corePKCS11 library is a reference implementation of the Public Key Cryptography Standard #11, to support provisioning and TLS client authentication. For more information, see the corePKCS11 library.
TLS		For more information, see Transport Layer Security.
Common I/O	Common I/O API Reference	For more information, see Common I/O.
Cellular Interface	Cellular Interface API Reference	The Cellular Interface library exposes the capabilities of a few popular cellular modems through a uniform API. For more information, see the Cellular Interface library.

FreeRTOS application libraries

You can optionally include the following standalone application libraries in your FreeRTOS configuration to interact with AWS IoT services on the cloud.

> **Note**
>
> Some of the application libraries have the same APIs as libraries in the AWS IoT Device SDK for Embedded C. For these libraries, see the AWS IoT Device SDK C API Reference. For more information about the AWS IoT Device SDK for Embedded C, see AWS IoT Device SDK for Embedded C.

FreeRTOS application libraries

Library	API Reference	Description
AWS IoT Device Defender	Device Defender C SDK API Reference	The FreeRTOS AWS IoT Device Defender library connects your FreeRTOS device to AWS IoT Device Defender. For more information, see AWS IoT Device Defender library.
AWS IoT Greengrass	Greengrass API Reference	The FreeRTOS AWS IoT Greengrass library connects your FreeRTOS device to AWS IoT Greengrass. For more information, see AWS IoT Greengrass Discovery library.
MQTT	MQTT (v1.x.x) Library API Reference MQTT (v1) Agent API Reference MQTT (v2.x.x) C SDK API Reference	The coreMQTT library provides a client for your FreeRTOS device to publish and subscribe to MQTT topics. MQTT is the protocol that devices use to interact with AWS IoT. For more information about the coreMQTT library version 3.0.0, see coreMQTT library.
coreMQTT Agent	coreMQTT Agent Library API Reference	The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library. It lets you create a dedicated MQTT agent task that manages an MQTT connection in the background and doesn't need any intervention from other tasks. The library provides thread safe equivalents to the

Library	API Reference	Description
		coreMQTT's APIs, so it can be used in multi-threaded environments. For more information about the coreMQTT Agent library see coreMQTT Agent library.
AWS IoT Device Shadow	Device Shadow C SDK API Reference	The AWS IoT Device Shadow library enables your FreeRTOS device to interact with AWS IoT device shadows. For more information, see AWS IoT Device Shadow library.

Configuring the FreeRTOS libraries

Configuration settings for FreeRTOS and the AWS IoT Device SDK for Embedded C are defined as C preprocessor constants. You can set configuration settings with a global configuration file, or by using a compiler option such as `-D` in `gcc`. Because configuration settings are defined as compile-time constants, a library must be rebuilt if a configuration setting is changed.

If you want to use a global configuration file to set configuration options, create and save the file with the name `iot_config.h`, and place it in your include path. Within the file, use `#define` directives to configure the FreeRTOS libraries, demos, and tests.

For more information about the supported global configuration options, see the Global Configuration File Reference.

backoffAlgorithm library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

The backoffAlgorithm library is a utility library that is used to space out repeated retransmissions of the same block of data, to avoid network congestion. This library calculates the backoff period for retrying network operations (like a failed network connection with the server) using an exponential backoff with jitter algorithm.

Exponential backoff with jitter is typically used when retrying a failed connection or network request to a server that is caused by network congestion or high loads on the server. It is used to spread out the timing of the retry requests created by multiple devices attempting network connections at the same time. In an environment with poor connectivity, a client can get disconnected at any time; so a backoff strategy also helps the client to conserve battery by not repeatedly attempting reconnections when they are unlikely to succeed.

The library is written in C and designed to be compliant with ISO C90 and MISRA C:2012. The library has no dependencies on any additional libraries other than the standard C library and has no heap allocation, making it suitable for IoT microcontrollers, but also fully portable to other platforms.

This library can be freely used and is distributed under the MIT open source license.

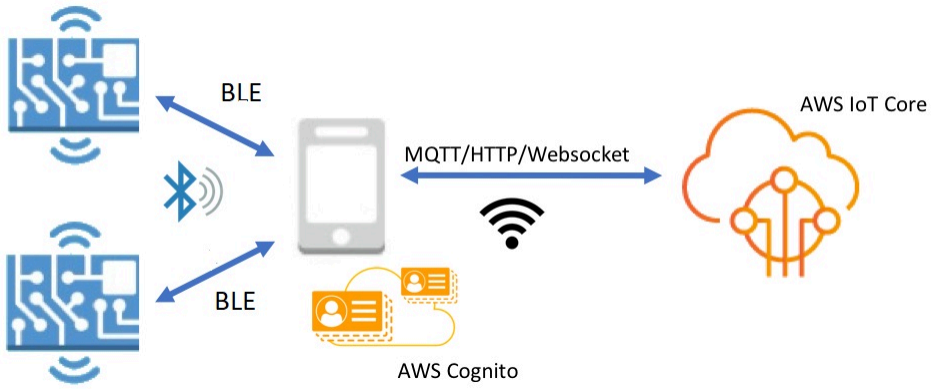
Code Size of backoffAlgorithm (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
backoff_algorithm.c	0.1K	0.1K
Total estimates	**0.1K**	**0.1K**

Bluetooth Low Energy library

> **Important**
>
> This library is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Overview

FreeRTOS supports publishing and subscribing to Message Queuing Telemetry Transport (MQTT) topics over Bluetooth Low Energy through a proxy device, such as a mobile phone. With the FreeRTOS Bluetooth Low Energy (BLE) library, your microcontroller can securely communicate with the AWS IoT MQTT broker.

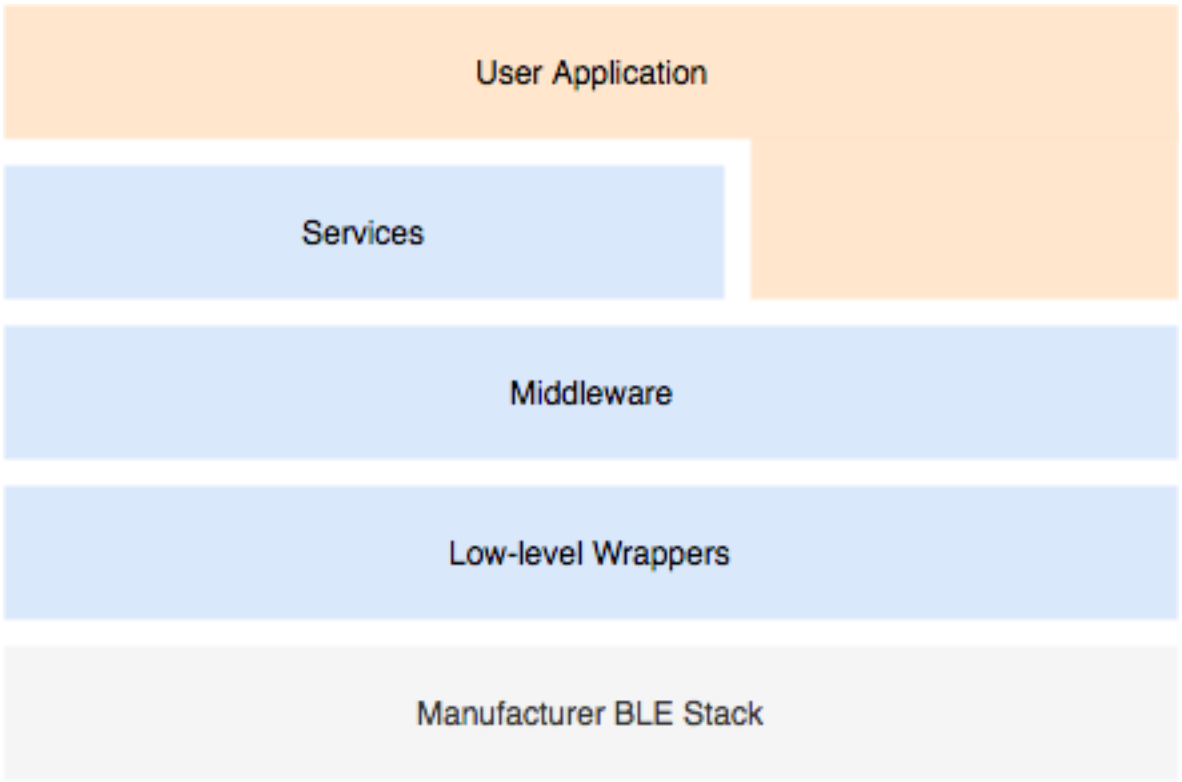


Using the Mobile SDKs for FreeRTOS Bluetooth Devices, you can write native mobile applications that communicate with the embedded applications on your microcontroller over BLE. For more information about the mobile SDKs, see Mobile SDKs for FreeRTOS Bluetooth devices.

The FreeRTOS BLE library includes services for configuring Wi-Fi networks, transferring large amounts of data, and providing network abstractions over BLE. The FreeRTOS BLE library also includes middleware and lower-level APIs for more direct control over your BLE stack.

Architecture

Three layers make up the FreeRTOS BLE library: services, middleware, and low-level wrappers.



Services

The FreeRTOS BLE services layer consists of four Generic Attribute (GATT) services that leverage the middleware APIs:

- Device information
- Wi-Fi provisioning
- Network abstraction
- Large object transfer

Device information

The Device information service gathers details about your microcontroller, including:

- The version of FreeRTOS that your device is using.
- The AWS IoT endpoint of the account for which the device is registered.
- Bluetooth Low Energy Maximum Transmission Unit (MTU).

Wi-Fi provisioning

The Wi-Fi provisioning service enables microcontrollers with Wi-Fi capabilities to do the following:

- List networks in range.
- Save networks and network credentials to flash memory.
- Set network priority.
- Delete networks and network credentials from flash memory.

Network abstraction

The network abstraction service abstracts the network connection type for applications. A common API interacts with your device's Wi-Fi, Ethernet, and Bluetooth Low Energy hardware stack, enabling an application to be compatible with multiple connection types.

Large Object Transfer

The Large Object Transfer service sends data to, and receives data from, a client. Other services, like Wi-Fi provisioning and Network abstraction, use the Large Object Transfer service to send and receive data. You can also use the Large Object Transfer API to directly interact with the service.

MQTT over BLE

MQTT over BLE contains the GATT profile for creating an MQTT proxy service over BLE. The MQTT proxy service allows an MQTT client to communicate with the AWS MQTT broker through a gateway device. For example, you can use the proxy service to connect a device running FreeRTOS to AWS MQTT through a smartphone app. The BLE device is the GATT server and exposes services and characteristics for the gateway device. The GATT server uses these exposed services and characteristics to perform MQTT operations with the cloud for that device. For more details, refer to Appendix A: MQTT over BLE GATT profile .

Middleware

FreeRTOS Bluetooth Low Energy middleware is an abstraction from the lower-level APIs. The middleware APIs make up a more user-friendly interface to the Bluetooth Low Energy stack.

Using middleware APIs, you can register several callbacks, across multiple layers, to a single event. Initializing the Bluetooth Low Energy middleware also initializes services and starts advertising.

Flexible callback subscription

Suppose your Bluetooth Low Energy hardware disconnects, and the MQTT over Bluetooth Low Energy service needs to detect the disconnection. An application that you wrote might also need to detect the same disconnection event. The Bluetooth Low Energy middleware can route the event to different parts of the code where you have registered callbacks, without making the higher layers compete for lower-level resources.

Low-level wrappers

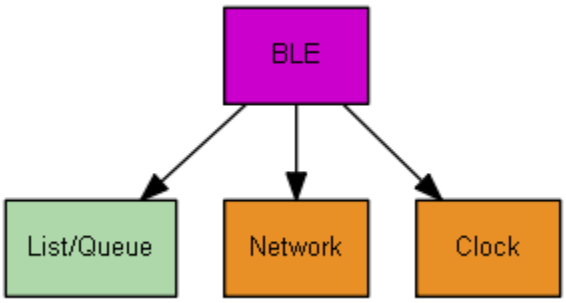
The low-level FreeRTOS Bluetooth Low Energy wrappers are an abstraction from the manufacturer's Bluetooth Low Energy stack. Low-level wrappers offer a common set of APIs for direct control over the hardware. The low-level APIs optimize RAM usage, but are limited in functionality.

Use the Bluetooth Low Energy service APIs to interact with the Bluetooth Low Energy services. The service APIs demand more resources than the low-level APIs.

Dependencies and requirements

The Bluetooth Low Energy library has the following direct dependencies:

- Linear Containers library
- A platform layer that interfaces with the operating system for thread management, timers, clock functions, and network access.



Only the Wi-Fi Provisioning service has FreeRTOS library dependencies:

GATT Service	Dependency
Wi-Fi Provisioning	Wi-Fi library

To communicate with the AWS IoT MQTT broker, you must have an AWS account and you must register your devices as AWS IoT things. For more information about setting up, see the AWS IoT Developer Guide.

FreeRTOS Bluetooth Low Energy uses Amazon Cognito for user authentication on your mobile device. To use MQTT proxy services, you must create an Amazon Cognito identity and user pools. Each Amazon Cognito Identity must have the appropriate policy attached to it. For more information, see the Amazon Cognito Developer Guide.

Library configuration file

Applications that use the FreeRTOS MQTT over Bluetooth Low Energy service must provide an `iot_ble_config.h` header file, in which configuration parameters are defined. Undefined configuration parameters take the default values specified in `iot_ble_config_defaults.h`.

Some important configuration parameters include:

`IOT_BLE_ADD_CUSTOM_SERVICES`

Allows users to create their own services.

`IOT_BLE_SET_CUSTOM_ADVERTISEMENT_MSG`

Allows users to customize the advertisement and scan response messages.

For more information, see Bluetooth Low Energy API Reference.

Optimization

When optimizing your board's performance, consider the following:

- Low-level APIs use less RAM, but offer limited functionality.
- You can set the `bleconfigMAX_NETWORK` parameter in the `iot_ble_config.h` header file to a lower value to reduce the amount of stack consumed.
- You can increase the MTU size to its maximum value to limit message buffering, and make code run faster and consume less RAM.

Usage restrictions

By default, the FreeRTOS Bluetooth Low Energy library sets the `eBTpropertySecureConnectionOnly` property to TRUE, which places the device in a Secure

Connections Only mode. As specified by the Bluetooth Core Specification v5.0, Vol 3, Part C, 10.2.4, when a device is in a Secure Connections Only mode, the highest LE security mode 1 level, level 4, is required for access to any attribute that has permissions higher than the lowest LE security mode 1 level, level 1. At the LE security mode 1 level 4, a device must have input and output capabilities for numeric comparison.

Here are the supported modes, and their associated properties:

Mode 1, Level 1 (No security)

```
/* Disable numeric comparison */
#define IOT_BLE_ENABLE_NUMERIC_COMPARISON        ( 0 )
#define IOT_BLE_ENABLE_SECURE_CONNECTION         ( 0 )
#define IOT_BLE_INPUT_OUTPUT                     ( eBTIONone )
#define IOT_BLE_ENCRYPTION_REQUIRED               ( 0 )
```

Mode 1, Level 2 (Unauthenticated pairing with encryption)

```
#define IOT_BLE_ENABLE_NUMERIC_COMPARISON        ( 0 )
#define IOT_BLE_ENABLE_SECURE_CONNECTION         ( 0 )
#define IOT_BLE_INPUT_OUTPUT                     ( eBTIONone )
```

Mode 1, Level 3 (Authenticated pairing with encryption)

This mode is not supported.

Mode 1, Level 4 (Authenticated LE Secure Connections pairing with encryption)

This mode is supported by default.

For information about LE security modes, see the Bluetooth Core Specification v5.0, Vol 3, Part C, 10.2.1.

Initialization

If your application interacts with the Bluetooth Low Energy stack through middleware, you only need to initialize the middleware. Middleware takes care of initializing the lower layers of the stack.

Middleware

To initialize the middleware

1. Initialize any Bluetooth Low Energy hardware drivers before you call the Bluetooth Low Energy middleware API.
2. Enable Bluetooth Low Energy.
3. Initialize the middleware with `IotBLE_Init()`.

> **Note**
>
> This initialization step is not required if you are running the AWS demos. Demo initialization is handled by the Network Manager, located at *`freertos`*`/demos/network_manager`.

Low-level APIs

If you don't want to use the FreeRTOS Bluetooth Low Energy GATT services, you can bypass the middleware and interact directly with the low-level APIs to save resources.

To initialize the low-level APIs

1. Initialize any Bluetooth Low Energy hardware drivers before you call the APIs. Driver initialization is not part of the Bluetooth Low Energy low-level APIs.
2. The Bluetooth Low Energy low-level API provides an enable/disable call to the Bluetooth Low Energy stack for optimizing power and resources. Before calling the APIs, you must enable Bluetooth Low Energy.

   ```
   const BTInterface_t * pxIface = BTGetBluetoothInterface();
   xStatus = pxIface->pxEnable( 0 );
   ```

3. The Bluetooth manager contains APIs that are common to both Bluetooth Low Energy and Bluetooth classic. The callbacks for the common manager must be initialized second.

   ```
   xStatus = xBTInterface.pxBTInterface->pxBtManagerInit( &xBTManagerCb );
   ```

4. The Bluetooth Low Energy adapter fits on top of the common API. You must initialize its callbacks like you initialized the common API.

```
xBTInterface.pxBTLeAdapterInterface = ( BTBleAdapter_t * )
 xBTInterface.pxBTInterface->pxGetLeAdapter();
xStatus = xBTInterface.pxBTLeAdapterInterface-
>pxBleAdapterInit( &xBTBleAdapterCb );
```

5. Register your new user application.

```
xBTInterface.pxBTLeAdapterInterface->pxRegisterBleApp( pxAppUuid );
```

6. Initialize the callbacks to the GATT servers.

```
xBTInterface.pxGattServerInterface = ( BTGattServerInterface_t * )
 xBTInterface.pxBTLeAdapterInterface->ppvGetGattServerInterface();
xBTInterface.pxGattServerInterface->pxGattServerInit( &xBTGattServerCb );
```

   After you initialize the Bluetooth Low Energy adapter, you can add a GATT server. You can register only one GATT server at a time.

```
xStatus = xBTInterface.pxGattServerInterface->pxRegisterServer( pxAppUuid );
```

7. Set application properties like secure connection only and MTU size.

```
xStatus = xBTInterface.pxBTInterface-
>pxSetDeviceProperty( &pxProperty[ usIndex ] );
```

API reference

For a full API reference, see Bluetooth Low Energy API Reference.

Example usage

The examples below demonstrate how to use the Bluetooth Low Energy library for advertising and creating new services. For full FreeRTOS Bluetooth Low Energy demo applications, see Bluetooth Low Energy Demo Applications.

Advertising

1. In your application, set the advertising UUID:

```
static const BTUuid_t _advUUID =
{
    .uu.uu128 = IOT_BLE_ADVERTISING_UUID,
    .ucType   = eBTuuidType128
};
```

2. Then define the `IotBle_SetCustomAdvCb` callback function:

```
void IotBle_SetCustomAdvCb( IotBleAdvertisementParams_t * pAdvParams,
 IotBleAdvertisementParams_t * pScanParams)
{
    memset(pAdvParams, 0, sizeof(IotBleAdvertisementParams_t));
    memset(pScanParams, 0, sizeof(IotBleAdvertisementParams_t));

    /* Set advertisement message */
    pAdvParams->pUUID1 = &_advUUID;
    pAdvParams->nameType = BTGattAdvNameNone;

    /* This is the scan response, set it back to true. */
    pScanParams->setScanRsp = true;
    pScanParams->nameType = BTGattAdvNameComplete;
}
```

   This callback sends the UUID in the advertisement message and the full name in the scan response.

3. Open `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/iot_ble_config.h`, and set `IOT_BLE_SET_CUSTOM_ADVERTISEMENT_MSG` to 1. This triggers the `IotBle_SetCustomAdvCb` callback.

Adding a new service

For full examples of services, see *`freertos`*`/.../ble/services`.

1. Create UUIDs for the service's characteristic and descriptors:

```
#define xServiceUUID_TYPE \
{\
    .uu.uu128 = gattDemoSVC_UUID, \
    .ucType   = eBTuuidType128 \
}
```

```
#define xCharCounterUUID_TYPE \
{\
    .uu.uu128 = gattDemoCHAR_COUNTER_UUID,\
    .ucType   = eBTuuidType128\
}
#define xCharControlUUID_TYPE \
{\
    .uu.uu128 = gattDemoCHAR_CONTROL_UUID,\
    .ucType   = eBTuuidType128\
}
#define xClientCharCfgUUID_TYPE \
{\
    .uu.uu16 = gattDemoCLIENT_CHAR_CFG_UUID,\
    .ucType  = eBTuuidType16\
}
```

2. Create a buffer to register the handles of the characteristic and descriptors:

```
static uint16_t usHandlesBuffer[egattDemoNbAttributes];
```

3. Create the attribute table. To save some RAM, define the table as a `const`.

> **Important**
>
> Always create the attributes in order, with the service as the first attribute.

```
static const BTAttribute_t pxAttributeTable[] = {
     {
         .xServiceUUID =  xServiceUUID_TYPE
     },
    {
         .xAttributeType = eBTDbCharacteristic,
         .xCharacteristic =
         {
              .xUuid = xCharCounterUUID_TYPE,
              .xPermissions = ( IOT_BLE_CHAR_READ_PERM ),
              .xProperties = ( eBTPropRead | eBTPropNotify )
          }
     },
     {
         .xAttributeType = eBTDbDescriptor,
```

```
          .xCharacteristicDescr =
          {
              .xUuid = xClientCharCfgUUID_TYPE,
              .xPermissions = ( IOT_BLE_CHAR_READ_PERM | IOT_BLE_CHAR_WRITE_PERM )
           }
      },
     {
          .xAttributeType = eBTDbCharacteristic,
          .xCharacteristic =
          {
               .xUuid = xCharControlUUID_TYPE,
               .xPermissions = ( IOT_BLE_CHAR_READ_PERM | IOT_BLE_CHAR_WRITE_PERM
  ),
               .xProperties = ( eBTPropRead | eBTPropWrite )
           }
      }
};
```

4. Create an array of callbacks. This array of callbacks must follow the same order as the table array defined above.

   For example, if `vReadCounter` gets triggered when `xCharCounterUUID_TYPE` is accessed, and `vWriteCommand` gets triggered when `xCharControlUUID_TYPE` is accessed, define the array as follows:

```
static const IotBleAttributeEventCallback_t pxCallBackArray[egattDemoNbAttributes]
 =
    {
  NULL,
  vReadCounter,
  vEnableNotification,
  vWriteCommand
};
```

5. Create the service:

```
static const BTService_t xGattDemoService =
{
  .xNumberOfAttributes = egattDemoNbAttributes,
  .ucInstId = 0,
  .xType = eBTServiceTypePrimary,
  .pusHandlesBuffer = usHandlesBuffer,
  .pxBLEAttributes = (BTAttribute_t *)pxAttributeTable
```

```
};
```

6. Call the API `IotBle_CreateService` with the structure that you created in the previous step. The middleware synchronizes the creation of all services, so any new services need to already be defined when the `IotBle_AddCustomServicesCb` callback is triggered.

   a. Set `IOT_BLE_ADD_CUSTOM_SERVICES` to 1 in `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/iot_ble_config.h`.

   b. Create IotBle_AddCustomServicesCb in your application:

```
void IotBle_AddCustomServicesCb(void)
{
    BTStatus_t xStatus;
    /* Select the handle buffer. */
    xStatus = IotBle_CreateService( (BTService_t *)&xGattDemoService,
 (IotBleAttributeEventCallback_t *)pxCallBackArray );
}
```

Porting

User input and output peripheral

A secure connection requires both input and output for numeric comparison. The `eBLENumericComparisonCallback` event can be registered using the event manager:

```
xEventCb.pxNumericComparisonCb = &prvNumericComparisonCb;
xStatus = BLE_RegisterEventCb( eBLENumericComparisonCallback, xEventCb );
```

The peripheral must display the numeric passkey and take the result of the comparison as an input.

Porting API implementations

To port FreeRTOS to a new target, you must implement some APIs for the Wi-Fi Provisioning service and Bluetooth Low Energy functionality.

Bluetooth Low Energy APIs

To use the FreeRTOS Bluetooth Low Energy middleware, you must implement some APIs.

APIs common between GAP for Bluetooth Classic and GAP for Bluetooth Low Energy

- `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`

APIs specific to GAP for Bluetooth Low Energy

- `pxRegisterBleApp`
- `pxUnregisterBleApp`
- `pxBleAdapterInit`
- `pxStartAdv`
- `pxStopAdv`
- `pxSetAdvData`
- `pxConnParameterUpdateRequest`
- `pxRegisterBleAdapterCb`
- `pxAdvStartCb`

- pxSetAdvDataCb
- pxConnParameterUpdateRequestCb
- pxCongestionCb

GATT server

- 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`

For more information about porting the FreeRTOS Bluetooth Low Energy library to your platform, see Porting the Bluetooth Low Energy Library in the FreeRTOS Porting Guide.

Mobile SDKs for FreeRTOS Bluetooth devices

> **Important**
>
> This library is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

You can use the Mobile SDKs for FreeRTOS Bluetooth Devices to create mobile applications that interact with your microcontroller over Bluetooth Low Energy. The Mobile SDKs can also communicate with AWS services, using Amazon Cognito for user authentication.

Android SDK for FreeRTOS Bluetooth devices

Use the Android SDK for FreeRTOS Bluetooth Devices to build Android mobile applications that interact with your microcontroller over Bluetooth Low Energy. The SDK is available on GitHub.

To install the Android SDK for FreeRTOS Bluetooth devices, follow the instructions for "Setting up the SDK" in the project's README.md file.

For information about setting up and running the demo mobile application that is included with the SDK, see Prerequisites and FreeRTOS Bluetooth Low Energy Mobile SDK demo application.

iOS SDK for FreeRTOS Bluetooth devices

Use the iOS SDK for FreeRTOS Bluetooth Devices to build iOS mobile applications that interact with your microcontroller over Bluetooth Low Energy. The SDK is available on GitHub.

To install the iOS SDK

1. Install CocoaPods:

```
$ gem install cocoapods
$ pod setup
```

> **Note**
>
> You might need to use sudo to install CocoaPods.

2. Install the SDK with CocoaPods (add this to your podfile):

```
$ pod 'FreeRTOS', :git => 'https://github.com/aws/amazon-freertos-ble-ios-sdk.git'
```

For information about setting up and running the demo mobile application that is included with the SDK, see Prerequisites and FreeRTOS Bluetooth Low Energy Mobile SDK demo application.

Appendix A: MQTT over BLE GATT profile

GATT Service Details

MQTT over BLE uses an instance of the data transfer GATT service to send MQTT Concise Binary Object Representation (CBOR) messages between the FreeRTOS device and the proxy device. The data transfer service exposes certain characteristics that help send and receive raw data over the BLE GATT protocol. It also handles the fragmentation and assembly of payloads greater than the BLE maximum transfer unit (MTU) size.

Service UUID

`A9D7-166A-D72E-40A9-A002-4804-4CC3-FF00`

Service Instances

One instance of the GATT service is created for each MQTT session with the broker. Each service has a unique UUID (two bytes) that identifies its type. Each individual instance is differentiated by the instance ID.

Each service is instantiated as a primary service on each BLE server device. You can create multiple instances of the service on a given device. The MQTT proxy service type has a unique UUID.

Characteristics

Characteristic content format: **CBOR**

Max characteristic value size : 512 bytes

Characteristic	Requirement	Mandatory Properties	Optional Properties	Security Permissions	Brief Description	UUID
Control	M	Write	None	Write Needs Encryption	Used to start and stop the MQTT proxy.	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF01
TXMessage	M	Read, Notification	None	Read Needs Encryption	Used to send a notification containing a message to a broker via a proxy.	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF02
RXMessage	M	Read, Write Without Response	None	Read, Write Needs Encryption	Used to receive a message from a broker via a proxy.	A9D7-166A-D72E-40A9-A002-48

Characteristic	Requirement	Mandatory Properties	Optional Properties	Security Permissions	Brief Description	UUID
						04-4CC3-FF03
TXLargeMessage	M	Read, Notification	None	Read Needs Encryption	Used to send a large message (Message > BLE MTU Size) to a broker via a proxy.	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF04
RXLargeMessage	M	Read, Write Without Response	None	Read, Write Needs Encryption	Used to receive large message (Message > BLE MTU Size) from a broker via a proxy.	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF05

GATT Procedure Requirements

Read Characteristic Values	Mandatory
Read Long Characteristic Values	Mandatory
Write Characteristic Values	Mandatory

Write Long Characteristic Values	Mandatory
Read Characteristic descriptors	Mandatory
Write Characteristic descriptors	Mandatory
Notifications	Mandatory
Indications	Mandatory

Message Types

The following message types are exchanged.

Message Type	Message	Map with these key / value pairs
0x01	CONNECT	• Key = "w", value = Type 0 Integer, Message type (1) • Key = "d", value = Type 3, Text String, Client Identifier for the session • Key = "a", value = Type 3, Text String, Broker endpoint for the session • Key = "c", Value = Simple Value Type True/False
0x02	CONNACK	• Key = "w, value = Type 0 Integer, Message type (2) • Key = "s", Value = Type 0 Integer, Status code
0x03	PUBLISH	• Key = "w", value = Type 0 Integer, Message type (3)

Message Type	Message	Map with these key / value pairs
		• Key = "u", value = Type 3, Text String, Topic for publish • Key = "n", value = Type 0, Integer, QoS for publish • Key = "i", value = Type 0, Integer, Message Identifier, Only for QoS 1 Publishes • Key = "k", Value = Type 2, Byte String, Payload for publish
0x04	PUBACK	• Sent Only for QoS 1 messages. • Key = "w", value = Type 0 Integer, Message type (4) • Key = "i", value = Type 0, Integer, Message Identifier
0x08	SUBSCRIBE	• Key = "w", value = Type 0 Integer, Message type (8) • Key = "v", value = Type 4, Array of text strings, topics for subscription • Key = "o", value = Type 4, Array of Integers, QoS for subscription • Key = "i", value = Type 0, Integer, Message Identifier

Message Type	Message	Map with these key / value pairs
0x09	SUBACK	• Key = "w", value = Type 0 Integer, Message type (9) • Key = "i", value = Type 0, Integer, Message Identifier • Key = "s", value = Type 0, Integer, Status code for Subscription
0X0A	UNSUBSCRIBE	• Key = "w", value = Type 0 Integer, Message type (10) • Key = "v", value = Type 4, Array of text strings, topics for unsubscription • Key = "i", value = Type 0, Integer, Message Identifier
0x0B	UNSUBACK	• Key = "w", value = Type 0 Integer, Message type (11) • Key = "i", value = Type 0, Integer, Message Identifier • Key = "s", value = Type 0, Integer, Status code for UnSubscription
0X0C	PINGREQ	• Key = "w", value = Type 0 Integer, Message type (12)
0x0D	PINGRESP	• Key = "w", value = Type 0 Integer, Message type (13)
0x0E	DISCONNNECT	• Key = "w", value = Type 0 Integer, Message type (14)

Large Payload Transfer Characteristics

TXLargeMessage

TXLargeMessage is used by the device to send a large payload that is greater than the MTU size negotiated for the BLE connection.

- The device sends the first MTU bytes of the payload as a notification through the characteristic.
- The proxy sends a read request on this characteristic for the remaining bytes.
- The device sends up to the MTU size or the remaining bytes of the payload, whichever is less. Each time, it increases the offset read by the size of the payload sent.
- The proxy will continue to read the characteristic until it gets a zero length payload or a payload less than the MTU size.
- If the device doesn't get a read request within a specified timeout, the transfer fails and the proxy and gateway release the buffer.
- If the proxy doesn't get a read response within a specified timeout, the transfer fails and the proxy releases the buffer.

RXLargeMessage

RXLargeMessage is used by the device to receive a large payload that is greater than the MTU size negotiated for the BLE connection.

- The proxy writes messages, up to the MTU size, one by one, using write with response on this characteristic.
- The device buffers the message until it receives a write request with zero length or a length less than the MTU size.
- If the device doesn't get a write request within a specified timeout, the transfer fails and the device releases the buffer.
- If the proxy doesn't get a write response within a specified timeout, the transfer fails and the proxy releases the buffer.

Cellular Interface library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

The Cellular Interface library implements a simple unified API that hides the complexity of the cellular modem-specific AT commands and exposes a socket-like interface to C programmers.

Most cellular modems implement more or less of the AT commands defined by the 3GPP TS v27.007 standard. This project provides an implementation of such standard AT commands in a reusable common component. The three Cellular Interface libraries in this project all take advantage of that common code. The library for each modem only implements the vendor-specific AT commands, then exposes the complete Cellular Interface library API.

The common component that implements the 3GPP TS v27.007 standard has been written in compliance with the following code quality criteria:

- GNU Complexity scores are not over 8
- MISRA C:2012 coding standard. Any deviations from the standard are documented in source code comments marked by "coverity".

Dependencies and requirements

There is no direct dependency for the Cellular Interface library. However, Ethernet, Wi-Fi and cellular cannot co-exist in the FreeRTOS network stack. Developers must choose one of the network interface to integrate with the Secure Sockets library.

Porting

For information about porting the Cellular Interface library to your platform, see Porting the Cellular Interface library in the *FreeRTOS Porting Guide*.

Memory use

Code Size of cellular interface library (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
cellular_3gpp_api.c	6.3K	5.7K
cellular_3gpp_urc_handler.c	0.9K	0.8K
cellular_at_core.c	1.4K	1.2K
cellular_common_api.c	0.5K	0.5K
cellular_common.c	1.6K	1.4K
cellular_pkthandler.c	1.4K	1.2K
cellular_pktio.c	1.8K	1.6K
Total estimates	**13.9K**	**12.4K**

Getting started

Download the source code

The source code can be downloaded as part of the FreeRTOS libraries or by itself.

To clone the library from Github using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Cellular-Interface.git
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Cellular-Interface.git
```

Folder structure

At the root of this repository you will see these folders:

- `source` : reusable common code that implements the standard AT commands defined by 3GPP TS v27.007
- `doc` : documentation
- `test` : unit test and cbmc
- `tools` : tools for Coverity static analysis and CMock

Configure and build the Library

The Cellular Interface library should be built as part of an application. In order to do this, you must provide certain configurations. The FreeRTOS_Cellular_Interface_Windows_Simulator project provides an example of how to configure the build. More information can be found in the Cellular API References.

Please refer to the Cellular Interface page for more information.

Integrate the Cellular Interface library with MCU platforms

The Cellular Interface library runs on MCUs using an abstracted interface, the Comm Interface, to communicate with cellular modems. A Comm Interface must be implemented on the MCU platform as well. The most common implementations of the Comm Interface communicate over UART hardware, but can be implemented over other physical interfaces, such as SPI, as well. The documentation for the Comm Interface can be found in the Cellular Library API References. The following example implementations of the Comm Interface are available:

- FreeRTOS Windows simulator comm interface
- FreeRTOS Common IO UART comm interface
- STM32 L475 discovery board comm interface
- Sierra Sensor Hub board comm interface

Common I/O

> **Important**
>
> This library is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an

existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Overview

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

FreeRTOS common I/O acts as this hardware abstraction layer. It provides a set of standard APIs for accessing common serial devices on supported reference boards. These common APIs communicate and interact with these peripherals and enable your code to function across platforms. Without common I/O, writing code to work with low level devices is silicon-vendor specific.

Supported peripherals

- UART
- SPI
- I2C

Supported features

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

Peripheral specific

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

Porting

For more information, see the FreeRTOS Porting Guide.

AWS IoT Device Defender library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

You can use the AWS IoT Device Defender library to send security metrics from your IoT devices to AWS IoT Device Defender. You can use AWS IoT Device Defender to continuously monitor these security metrics from devices for deviations from what you have defined as appropriate behavior for each device. If something doesn't look right, AWS IoT Device Defender sends out an alert so that you can take action to fix the issue. Interactions with AWS IoT Device Defender use MQTT, a lightweight publish-subscribe protocol. This library provides an API to compose and recognize the MQTT topic strings used by AWS IoT Device Defender.

For more information, see AWS IoT Device Defender in the *AWS IoT Developer Guide*.

The library is written in C and designed to be compliant with ISO C90 and MISRA C:2012. The library has no dependencies on any additional libraries other than the standard C library. It also doesn't have any platform dependencies, such as threading or synchronization. It can be used with any MQTT library and any JSON or CBOR library. The library has proofs showing safe memory use and no heap allocation, making it suitable for IoT microcontrollers, but also fully portable to other platforms.

The AWS IoT Device Defender library can be freely used and is distributed under the MIT open source license.

Code Size of AWS IoT Device Defender (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
defender.c	1.1K	0.6K

Code Size of AWS IoT Device Defender (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
Total estimates	**1.1K**	**0.6K**

AWS IoT Greengrass Discovery library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Overview

The AWS IoT Greengrass Discovery library is used by your microcontroller devices to discover a Greengrass core on your network. Using the AWS IoT Greengrass Discovery APIs, your device can send messages to a Greengrass core after it finds the core's endpoint.

Dependencies and requirements

To use the Greengrass Discovery library, you must create a thing in AWS IoT, including a certificate and policy. For more information, see AWS IoT Getting Started.

You must set values for the following constants in the *freertos*`/demos/include/aws_clientcredential.h` file:

`clientcredentialMQTT_BROKER_ENDPOINT`

Your AWS IoT endpoint.

`clientcredentialIOT_THING_NAME`

The name of your IoT thing.

`clientcredentialWIFI_SSID`

The SSID for your Wi-Fi network.

`clientcredentialWIFI_PASSWORD`

Your Wi-Fi password.

`clientcredentialWIFI_SECURITY`

The type of security used by your Wi-Fi network.

You must also set values for the following constants in the *`freertos`*`/demos/include/aws_clientcredential_keys.h` file:

`keyCLIENT_CERTIFICATE_PEM`

The certificate PEM associated with your thing.

`keyCLIENT_PRIVATE_KEY_PEM`

The private key PEM associated with your thing.

You must have a Greengrass group and core device set up in the console. For more information, see Getting Started with AWS IoT Greengrass.

Although the coreMQTT library is not required for Greengrass connectivity, we strongly recommend you install it. The library can be used to communicate with the Greengrass core after it has been discovered.

API reference

For a full API reference, see Greengrass API Reference.

Example usage

Greengrass workflow

The MCU device initiates the discovery process by requesting from AWS IoT a JSON file that contains the Greengrass core connectivity parameters. There are two methods for retrieving the Greengrass core connectivity parameters from the JSON file:

- Automatic selection iterates through all of the Greengrass cores listed in the JSON file and connects to the first one available.
- Manual selection uses the information in `aws_ggd_config.h` to connect to the specified Greengrass core.

How to use the Greengrass API

All default configuration options for the Greengrass API are defined in `aws_ggd_config_defaults.h`.

If only one Greengrass core is present, call `GGD_GetGGCIPandCertificate` to request the JSON file with Greengrass core connectivity information. When `GGD_GetGGCIPandCertificate` is returned, the `pcBuffer` parameter contains the text of the JSON file. The `pxHostAddressData` parameter contains the IP address and port of the Greengrass core to which you can connect.

For more customization options, like dynamically allocating certificates, you must call the following APIs:

`GGD_JSONRequestStart`

Makes an HTTP GET request to AWS IoT to initiate the discovery request to discover a Greengrass core. `GD_SecureConnect_Send` is used to send the request to AWS IoT.

`GGD_JSONRequestGetSize`

Gets the size of the JSON file from the HTTP response.

`GGD_JSONRequestGetFile`

Gets the JSON object string. `GGD_JSONRequestGetSize` and `GGD_JSONRequestGetFile` use `GGD_SecureConnect_Read` to get the JSON data from the socket. `GGD_JSONRequestStart`, `GGD_SecureConnect_Send`, `GGD_JSONRequestGetSize` must be called to receive the JSON data from AWS IoT.

`GGD_GetIPandCertificateFromJSON`

Extracts the IP address and the Greengrass core certificate from the JSON data. You can turn on automatic selection by setting the `xAutoSelectFlag` to `True`. Automatic selection finds the first core device your FreeRTOS device can connect to. To connect to a Greengrass core, call the `GGD_SecureConnect_Connect` function, passing in the IP address, port, and certificate of the core device. To use manual selection, set the following fields of the `HostParameters_t` parameter:

`pcGroupName`

The ID of the Greengrass group to which the core belongs. You can use the `aws greengrass list-groups` CLI command to find the ID of your Greengrass groups.

`pcCoreAddress`

The ARN of the Greengrass core to which you are connecting.

coreHTTP library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

HTTP C client library for small IoT devices (MCU or small MPU)

Introduction

The coreHTTP library is a client implementation of a subset of the HTTP/1.1 standard. The HTTP standard provides a stateless protocol that runs on top of TCP/IP and is often used in distributed, collaborative, hypertext information systems.

The coreHTTP library implements a subset of the HTTP/1.1 protocol standard. This library has been optimized for a low memory footprint. The library provides a fully synchronous API so applications can completely manage their concurrency. It uses fixed buffers only, so that applications have complete control of their memory allocation strategy.

The library is written in C and designed to be compliant with ISO C90 and MISRA C:2012. The library's only dependencies are the standard C library and LTS version (v12.19.1) of the http-parser from Node.js. The library has proofs showing safe memory use and no heap allocation, making it suitable for IoT microcontrollers, but also fully portable to other platforms.

When using HTTP connections in IoT applications, we recommend that you use a secure transport interface, such as one that uses the TLS protocol as demonstrated in the coreHTTP mutual authentication demo.

This library can be freely used and is distributed under the MIT open source license.

Code Size of coreHTTP (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
core_http_client.c	3.2K	2.6K
api.c (llhttp)	2.6K	2.0K
http.c (llhttp)	0.3K	0.3K
llhttp.c (llhttp)	17.9	15.9
Total estimates	**23.9K**	**20.7K**

coreJSON library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

JSON (JavaScript Object Notation) is a human-readable data serialization format. It is widely used to exchange data, such as with the AWS IoT Device Shadow service, and is part of many APIs, such as the GitHub REST API. JSON is maintained as a standard by Ecma International.

The coreJSON library provides a parser that supports key lookups while strictly enforcing the ECMA-404 Standard JSON Data Interchange syntax. The library is written in C and designed to comply with ISO C90 and MISRA C:2012. It has proofs showing safe memory use and no heap allocation, making it suitable for IoT microcontrollers, but also fully portable to other platforms.

Memory use

The coreJSON library uses an internal stack to track nested structures in a JSON document. The stack exists for the duration of a single function call; it is not preserved. Stack size may be specified by defining the macro, `JSON_MAX_DEPTH`, which defaults to 32 levels. Each level consumes a single byte.

Code Size of coreJSON (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
core_json.c	2.9K	2.4K
Total estimates	**2.9K**	**2.4K**

coreMQTT library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

The coreMQTT library is a client implementation of the MQTT (Message Queue Telemetry Transport) standard. The MQTT standard provides a lightweight publish/subscribe (or PubSub) messaging protocol that runs on top of TCP/IP and is often used in Machine to Machine (M2M) and Internet of Things (IoT) use cases.

The coreMQTT library is compliant with the MQTT 3.1.1 protocol standard. This library has been optimized for a low memory footprint. The design of this library embraces different use-cases, ranging from resource-constrained platforms using only QoS 0 MQTT PUBLISH messages to resource-rich platforms using QoS 2 MQTT PUBLISH over TLS (Transport Layer Security) connections. The library provides a menu of composable functions, which can be chosen and combined to precisely fit the needs of a particular use-case.

The library is written in **C** and designed to be compliant with ISO C90 and MISRA C:2012. This MQTT library has no dependencies on any additional libraries except for the following:

- The standard C library
- A customer-implemented network transport interface
- (Optional) A user-implemented platform time function

The library is decoupled from the underlying network drivers through the provision of a simple send and receive transport interface specification. The application writer can select an existing transport interface, or implement their own as appropriate for their application.

The library provides a high-level API to connect to an MQTT broker, subscribe/unsubscribe to a topic, publish a message to a topic and receive incoming messages. This API takes the transport interface described above as a parameter and uses that to send and receive messages to and from the MQTT broker.

The library also exposes low level serializer/deserializer API. This API can be used to build a simple IoT application consisting of only the required a subset of MQTT functionality, without any other overhead. The serializer/deserializer API can be used in conjunction with any available transport layer API, like sockets, to send and receive messages to and from the broker.

When using MQTT connections in IoT applications, we recommended that you use a secure transport interface, such as one that uses the TLS protocol.

This MQTT library doesn't have platform dependencies, such as threading or synchronization. This library does have proofs that demonstrate safe memory use and no heap allocation, which makes it suitable for IoT microcontrollers, but also fully portable to other platforms. It can be freely used, and is distributed under the MIT open source license.

Code Size of coreMQTT (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
core_mqtt.c	4.0K	3.4K
core_mqtt_state.c	1.7K	1.3K
core_mqtt_serializer.c	2.8K	2.2K
Total estimates	**8.5K**	**6.9K**

coreMQTT Agent library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library. It lets you create a dedicated MQTT agent task that manages an MQTT connection in the background and doesn't need any intervention from other tasks. The library provides thread safe equivalents to the coreMQTT's APIs, so it can be used in multi-threaded environments.

The MQTT agent is an independent task (or thread of execution). It achieves thread safety by being the only task that is permitted to access the MQTT library's API. It serializes access by isolating all MQTT API calls to a single task, and it removes the need for semaphores or any other synchronization primitives.

The library uses a thread safe messaging queue (or other inter-process communication mechanism) to serialize all requests to call MQTT APIs. The messaging implementation is decoupled from the library through a messaging interface, which allows the library to be ported to other operating systems. The messaging interface is composed of functions to send and receive pointers to the agent's command structures, and functions to allocate these command objects, which allows the application writer to decide the memory allocation strategy appropriate for their application.

The library is written in C and designed to be compliant with ISO C90 and MISRA C:2012. The library has no dependencies on any additional libraries other than coreMQTT library and the standard C library. The library has proofs that show safe memory use and no heap allocation, so it can be used for IoT microcontrollers, but is also fully portable to other platforms.

This library can be freely used and is distributed under the MIT open source license.

Code Size of coreMQTT Agent (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
core_mqtt_agent.c	1.7K	1.5K

Code Size of coreMQTT Agent (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
core_mqtt_agent_command_functions.c	0.3K	0.2K
core_mqtt.c (coreMQTT)	4.0K	3.4K
core_mqtt_state.c (coreMQTT)	1.7K	1.3K
core_mqtt_serializer.c (coreMQTT)	2.8K	2.2K
Total estimates	**10.5K**	**8.6K**

AWS IoT Over the air (OTA) library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

The AWS IoT Over-the-air (OTA) update library enables you to manage the notification, download, and verification of firmware updates for FreeRTOS devices using HTTP or MQTT as the protocol. By using the OTA Agent library, you can logically separate firmware updates and the application running on your devices. The OTA Agent can share a network connection with the application. By sharing a network connection, you can potentially save a significant amount of RAM. In addition, the OTA Agent library lets you define application-specific logic for testing, committing, or rolling back a firmware update.

The Internet of Things (IoT) extends internet connectivity to embedded devices that were traditionally not connected. These devices can be programmed to communicate usable data over the internet, and can be remotely monitored and controlled. With advances in technology, these traditional embedded devices are getting internet capabilities in consumer, industrial, and enterprise spaces at a fast pace.

IoT devices are typically deployed in large quantities and often in places that are difficult or impractical for a human operator to access. Imagine a scenario where a security vulnerability that can expose data is discovered. In such scenarios, it is important to update the affected devices with security fixes quickly and reliably. Without the ability to perform OTA updates, it can also be difficult to update devices that are geographically dispersed. Having a technician update these devices will be costly, time consuming, and often times impractical. The time required to update these devices leaves them exposed to security vulnerabilities for a longer period. Recalling these devices for updating will also be costly and may cause significant disruption to consumers due to downtime.

Over the Air (OTA) Updates make it possible to update device firmware without an expensive recall or technician visit. This method adds the following benefits:

- **Security** - The ability to quickly respond to security vulnerabilities and software bugs that are discovered after the devices are deployed in the field.
- **Innovation** - Products can be updated frequently as new features are developed, driving the innovation cycle. The updates can take effect quickly with minimum downtime compared to traditional update methods.
- **Cost** - OTA updates can reduce maintenance costs significantly compared to methods traditionally used to update these devices.

Providing the OTA functionality requires the following design considerations:

- **Secure Communication** - Updates must use encrypted communication channels to prevent the downloads from being tampered with during transit.
- **Recovery** - Updates can fail due to things like intermittent network connectivity or receiving an invalid update. In these scenarios, the device needs to be able to return to a stable state and avoid becoming bricked.
- **Author Verification** - Updates must be verified to be from a trusted source, along with other validations like checking the version and compatibility.

For more information about setting up OTA updates with FreeRTOS, see FreeRTOS Over-the-Air Updates.

AWS IoT Over the air (OTA) library

The AWS IoT OTA library enables you to manage notifications of a newly available updates, download them, and perform cryptographic verification of firmware updates. Using the over-the-air (OTA) client library, you can logically separate the firmware update mechanisms from the application running on your device. The over-the-air (OTA) client library can share a network connection with the application, saving memory in resource-constrained devices. In addition, the over-the-air (OTA) client library lets you define application-specific logic for testing, committing, or rolling back a firmware update. The library supports different application protocols like Message Queuing Telemetry Transport (MQTT) and Hypertext Transfer Protocol (HTTP) and provides various configuration options you can fine tune for your network type and conditions.

This library's APIs provide these major functions:

- Register for notifications or poll for new update requests that are available.
- Receive, parse and validate the update request.
- Download and verify the file according to the information in the update request.
- Run a self-test before activating the received update to ensure the functional validity of the update.
- Update the status of the device.

This library uses AWS services to manage various cloud related functions such as sending firmware updates, monitoring large numbers of devices across multiple regions, reducing the blast radius of faulty deployments, and verifying the security of updates. This library can be used with any MQTT or HTTP library.

The demos for this library demonstrate complete over-the-air updates using the coreMQTT Library and AWS Services on a FreeRTOS device.

Features

Here is the complete OTA Agent interface:

`OTA_Init`

Initializes the OTA engine by starting OTA Agent ("OTA Task") in the system. Only one OTA Agent may exist.

OTA_Shutdown

Signal to the OTA Agent to shut down. The OTA agent will optionally unsubscribe from all MQTT job notification topics, stop in progress OTA jobs, if any, and clear all resources.

OTA_GetState

Gets the current state of the OTA Agent.

OTA_ActivateNewImage

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

OTA_SetImageState

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

OTA_GetImageState

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

OTA_CheckForUpdate

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

OTA_Suspend

Suspend all OTA Agent operations.

OTA_Resume

Resume OTA Agent operations.

OTA_SignalEvent

Signal an event to the OTA Agent task.

OTA_EventProcessingTask

OTA agent event processing loop.

OTA_GetStatistics

Get the statistics of OTA message packets which includes the number of packets received, queued, processed and dropped.

`OTA_Err_strerror`

Error code to string conversion for OTA errors.

`OTA_JobParse_strerror`

Convert an OTA Job Parsing error code to a string.

`OTA_PalStatus_strerror`

Status code to string conversion for OTA PAL status.

`OTA_OsStatus_strerror`

Status code to string conversion for OTA OS status.

API reference

For more information, see the AWS IoT Over-the-air Update: Functions.

Example usage

A typical OTA-capable device application using the MQTT protocol drives the OTA Agent by using the following sequence of API calls.

1. Connect to the AWS IoT coreMQTT Agent. For more information, see coreMQTT Agent library.
2. Initialize the OTA Agent by calling `OTA_Init`, including the buffers, the required ota interfaces, the thing name and the application callback. The callback implements application-specific logic that executes after completing an OTA update job.
3. When the OTA update is complete, FreeRTOS calls the job completion callback with one of the following events: `accepted`, `rejected`, or `self test`.
4. If the new firmware image has been rejected (for example, due to a validation error), the application can typically ignore the notification and wait for the next update.
5. If the update is valid and has been marked as accepted, call `OTA_ActivateNewImage` to reset the device and boot the new firmware image.

Porting

For information about porting OTA functionality to your platform, see Porting the OTA Library in the FreeRTOS Porting Guide.

Memory use

Code Size of AWS IoT OTA (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
ota.c	8.3K	7.5K
ota_interface.c	0.1K	0.1K
ota_base64.c	0.6K	0.6K
ota_mqtt.c	2.4K	2.2K
ota_cbor.c	0.8K	0.6K
ota_http.c	0.3K	0.3K
Total estimates	**12.5K**	**11.3K**

corePKCS11 library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Overview

The Public Key Cryptography Standard #11 defines a platform-independent API to manage and use cryptographic tokens. PKCS #11 refers to the API defined by the standard and to the standard itself. 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, and it's important because the functions it specifies allow application software to use, create, modify, and delete cryptographic objects, without ever exposing those objects to the application's memory. For example, FreeRTOS AWS reference integrations use a small subset of the PKCS #11 API to access the secret (private) key necessary to create a network connection that is

authenticated and secured by the Transport Layer Security (TLS) protocol without the application ever 'seeing' the key.

The corePKCS11 library contains a software-based mock implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. Using a software mock enables rapid development and flexibility, but it's expected that you will replace the mock with an implementation specific to the secure key storage used in your production devices. Generally, vendors for secure cryptoprocessors, such as Trusted Platform Module (TPM), Hardware Security Module (HSM), Secure Element, or any other type of secure hardware enclave, distribute a PKCS #11 implementation with the hardware. The purpose of the corePKCS11 software only mock library is therefore to provide a non hardware specific PKCS #11 implementation that allows for rapid prototyping and development before switching to a cryptoprocessor specific PKCS #11 implementation in production devices.

Only a subset of the PKCS #11 standard is implemented, with a focus on operations involving asymmetric keys, random number generation, and hashing. The targeted use cases include certificate and key management for TLS authentication, and code-sign signature verification, on small embedded devices. See the file `pkcs11.h` (obtained from OASIS, the standard body) in the FreeRTOS source code repository. In the FreeRTOS reference implementation, PKCS #11 API calls are made by the TLS helper interface in order to perform TLS client authentication during `SOCKETS_Connect`. PKCS #11 API calls are also made by our one-time developer provisioning workflow to import a TLS client certificate and private key for authentication to the AWS IoT MQTT broker. Those two use cases, provisioning and TLS client authentication, require implementation of only a small subset of the PKCS #11 interface standard.

Features

The following subset of PKCS #11 is used. This list is in roughly the order in which the routines are called in support of provisioning, TLS client authentication, and cleanup. For detailed descriptions of the functions, see the PKCS #11 documentation provided by the standard committee.

General setup and tear down API

- `C_Initialize`
- `C_Finalize`
- `C_GetFunctionList`
- `C_GetSlotList`
- `C_GetTokenInfo`

- `C_OpenSession`
- `C_CloseSession`
- `C_Login`

Provisioning API

- `C_CreateObject CKO_PRIVATE_KEY` (for device private key)
- `C_CreateObject CKO_CERTIFICATE` (for device certificate and code verification certificate)
- `C_GenerateKeyPair`
- `C_DestroyObject`

Client authentication

- `C_GetAttributeValue`
- `C_FindObjectsInit`
- `C_FindObjects`
- `C_FindObjectsFinal`
- `C_GenerateRandom`
- `C_SignInit`
- `C_Sign`
- `C_VerifyInit`
- `C_Verify`
- `C_DigestInit`
- `C_DigestUpdate`
- `C_DigestFinal`

Asymmetric cryptosystem support

The FreeRTOS reference implementation uses PKCS #11 2048-bit RSA (signing only) and ECDSA with the NIST P-256 curve. The following instructions describe how to create an AWS IoT thing based on a P-256 client certificate.

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

```
aws --version
aws-cli/1.11.176 Python/2.7.9 Windows/8 botocore/1.7.34

openssl version
OpenSSL 1.0.2g  1 Mar 2016
```

The following procedure assumes that you used the `aws configure` command to configure the AWS CLI. For more information, see Quick configuration with `aws configure` in the *AWS Command Line Interface User Guide*.

To create an AWS IoT thing based on a P-256 client certificate

1. Create an AWS IoT thing.

   ```
   aws iot create-thing --thing-name thing-name
   ```

2. Use OpenSSL to create a P-256 key.

   ```
   openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
    ec_param_enc:named_curve -outform PEM -out thing-name.key
   ```

3. Create a certificate enrollment request signed by the key created in step 2.

   ```
   openssl req -new -nodes -days 365 -key thing-name.key -out thing-name.req
   ```

4. Submit the certificate enrollment request to AWS IoT.

   ```
   aws iot create-certificate-from-csr  \
     --certificate-signing-request file://thing-name.req --set-as-active  \
     --certificate-pem-outfile thing-name.crt
   ```

5. Attach the certificate (referenced by the ARN output by the previous command) to the thing.

   ```
   aws iot attach-thing-principal --thing-name thing-name \
     --principal "arn:aws:iot:us-
   east-1:123456789012:cert/
   86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de729"
   ```

6. Create a policy. (This policy is too permissive. It should be used for development purposes only.)

```
aws iot create-policy --policy-name FullControl --policy-document file://
policy.json
```

The following is a listing of the policy.json file specified in the `create-policy` command. You can omit the `greengrass:*` action if you don't want to run the FreeRTOS demo for Greengrass connectivity and discovery.

JSON

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

7. Attach the principal (certificate) and policy to the thing.

```
aws iot attach-principal-policy --policy-name FullControl \
  --principal "arn:aws:iot:us-
east-1:123456789012:cert/
86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de729"
```

Now, follow the steps in the AWS IoT Getting Started section of this guide. Don't forget to copy the certificate and private key you created into your `aws_clientcredential_keys.h` file. Copy your thing name into `aws_clientcredential.h`.

Note

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

Porting

For information about porting the corePKCS11 library to your platform, see Porting the corePKCS11 Library in the FreeRTOS Porting Guide.

Memory use

Code Size of corePKCS11 (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
core_pkcs11.c	0.8K	0.8K
core_pki_utils.c	0.5K	0.3K
core_pkcs11_mbedtls.c	8.9K	7.5K
Total estimates	**10.2K**	**8.6K**

Secure Sockets library

Important

This library is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Overview

You can use the FreeRTOS Secure Sockets library to create embedded applications that communicate securely. The library is designed to make onboarding easy for software developers from various network programming backgrounds.

The FreeRTOS Secure Sockets library is based on the Berkeley sockets interface, with an additional secure communication option by TLS protocol. For information about the differences between the FreeRTOS Secure Sockets library and the Berkeley sockets interface, see `SOCKETS_SetSockOpt` in the Secure Sockets API Reference.

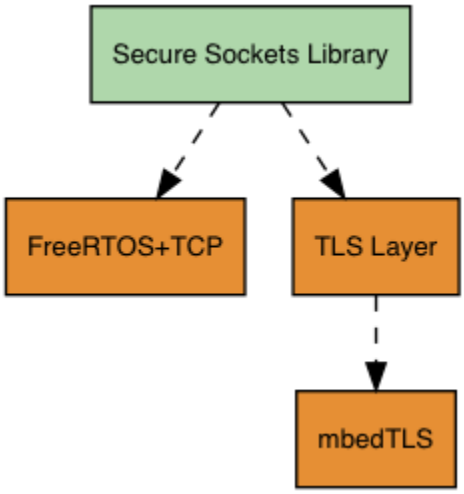
> **Note**
>
> Currently, only client APIs, plus a lightweight IP (lwIP) implementation of the server side `Bind` API, are supported for FreeRTOS Secure Sockets.

Dependencies and requirements

The FreeRTOS Secure Sockets library depends on a TCP/IP stack and on a TLS implementation. Ports for FreeRTOS meet these dependencies in one of three ways:

- A custom implementation of both TCP/IP and TLS
- A custom implementation of TCP/IP, and the FreeRTOS TLS layer with mbedTLS
- FreeRTOS+TCP and the FreeRTOS TLS layer with mbedTLS

The dependency diagram below shows the reference implementation included with the FreeRTOS Secure Sockets library. This reference implementation supports TLS and TCP/IP over Ethernet and Wi-Fi with FreeRTOS+TCP and mbedTLS as dependencies. For more information about the FreeRTOS TLS layer, see Transport Layer Security.



Features

FreeRTOS Secure Sockets library features include:

- A standard, Berkeley Sockets-based interface
- Thread-safe APIs for sending and receiving data
- Easy-to-enable TLS

Troubleshooting

Error codes

The error codes that the FreeRTOS Secure Sockets library returns are negative values. For more information about each error code, see Secure Sockets Error Codes in the Secure Sockets API Reference.

> **Note**
>
> If the FreeRTOS Secure Sockets API returns an error code, the coreMQTT library, which depends on the FreeRTOS Secure Sockets library, returns the error code `AWS_IOT_MQTT_SEND_ERROR`.

Developer support

The FreeRTOS Secure Sockets library includes two helper macros for handling IP addresses:

`SOCKETS_inet_addr_quick`

This macro converts an IP address that is expressed as four separate numeric octets into an IP address that is expressed as a 32-bit number in network-byte order.

`SOCKETS_inet_ntoa`

This macro converts an IP address that is expressed as a 32-bit number in network byte order to a string in decimal-dot notation.

Usage restrictions

Only TCP sockets are supported by the FreeRTOS Secure Sockets library. UDP sockets are not supported.

Server APIs are not supported by the FreeRTOS Secure Sockets library, except for a lightweight IP (lwIP) implementation of the server side `Bind` API. Client APIs are supported.

Initialization

To use the FreeRTOS Secure Sockets library, you need to initialize the library and its dependencies. To initialize the Secure Sockets library, use the following code in your application:

```
BaseType_t xResult = pdPASS;
xResult = SOCKETS_Init();
```

Dependent libraries must be initialized separately. For example, if FreeRTOS+TCP is a dependency, you need to invoke `FreeRTOS_IPInit` in your application as well.

API reference

For a full API reference, see Secure Sockets API Reference.

Example usage

The following code connects a client to a server.

```
#include "aws_secure_sockets.h"

#define configSERVER_ADDR0                    127
#define configSERVER_ADDR1                    0
#define configSERVER_ADDR2                    0
#define configSERVER_ADDR3                    1
```

```
#define configCLIENT_PORT                       443

/* Rx and Tx timeouts are used to ensure the sockets do not wait too long for
 * missing data. */
static const TickType_t xReceiveTimeOut = pdMS_TO_TICKS( 2000 );
static const TickType_t xSendTimeOut = pdMS_TO_TICKS( 2000 );

/* PEM-encoded server certificate */
/* The certificate used below is one of the Amazon Root CAs.\
Change this to the certificate of your choice. */
static const char cTlsECHO_SERVER_CERTIFICATE_PEM[] =
"-----BEGIN CERTIFICATE-----\n"
"MIIBtjCCAVugAwIBAgITBmyf1XSXNmY/Owua2eiedgPySjAKBggqhkjOPQQDAjA5\n"
"MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6b24g\n"
"Um9vdCBDQSAzMB4XDTE1MDUyNjAwMDAwMFoXDTQwMDUyNjAwMDAwMFowOTELMAkG\n"
"A1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJvb3Qg\n"
"Q0EgMzBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCmXp8ZBf8ANm+gBG1bG8lKl\n"
"ui2yEujSLtf6ycXYqm0fc4E7O5hrOXwzpcVOho6AF2hiRVd9RFgdszflZwjrZt6j\n"
"QjBAMA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/BAQDAgGGMB0GA1UdDgQWBBSr\n"
"ttvXBp43rDCGB5Fwx5zEGbF4wDAKBggqhkjOPQQDAgNJADBGAiEA4IWSoxe3jfkr\n"
"BqWTrBqYaGFy+uGh0PsceGCmQ5nFuMQCIQCcAu/xlJyzlvnrxir4tiz+OpAUFteM\n"
"YyRIHN8wfdVoOw==\n"
"-----END CERTIFICATE-----\n";

static const uint32_t ulTlsECHO_SERVER_CERTIFICATE_LENGTH =
 sizeof( cTlsECHO_SERVER_CERTIFICATE_PEM );

void vConnectToServerWithSecureSocket( void )
{
    Socket_t xSocket;
    SocketsSockaddr_t xEchoServerAddress;
    BaseType_t xTransmitted, lStringLength;

    xEchoServerAddress.usPort = SOCKETS_htons( configCLIENT_PORT );
    xEchoServerAddress.ulAddress = SOCKETS_inet_addr_quick( configSERVER_ADDR0,
                                                            configSERVER_ADDR1,
                                                            configSERVER_ADDR2,
                                                            configSERVER_ADDR3 );

    /* Create a TCP socket. */
    xSocket = SOCKETS_Socket( SOCKETS_AF_INET, SOCKETS_SOCK_STREAM,
 SOCKETS_IPPROTO_TCP );
    configASSERT( xSocket != SOCKETS_INVALID_SOCKET );
```

```
    /* Set a timeout so a missing reply does not cause the task to block indefinitely.
 */
    SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_RCVTIMEO, &xReceiveTimeOut,
 sizeof( xReceiveTimeOut ) );
    SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_SNDTIMEO, &xSendTimeOut,
 sizeof( xSendTimeOut ) );

    /* Set the socket to use TLS. */
    SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_REQUIRE_TLS, NULL, ( size_t ) 0 );
    SOCKETS_SetSockOpt( xSocket, 0, SOCKETS_SO_TRUSTED_SERVER_CERTIFICATE,
 cTlsECHO_SERVER_CERTIFICATE_PEM, ulTlsECHO_SERVER_CERTIFICATE_LENGTH );

    if( SOCKETS_Connect( xSocket, &xEchoServerAddress, sizeof( xEchoServerAddress ) )
 == 0 )
    {
        /* Send the string to the socket. */
        xTransmitted = SOCKETS_Send( xSocket,                         /* The socket
 receiving. */
                                     ( void * )"some message",        /* The data being
 sent. */
                                     12,                              /* The length of
 the data being sent. */
                                     0 );                             /* No flags. */

        if( xTransmitted < 0 )
        {
            /* Error while sending data */
            return;
        }

        SOCKETS_Shutdown( xSocket, SOCKETS_SHUT_RDWR );
    }
    else
    {
        //failed to connect to server
    }

    SOCKETS_Close( xSocket );
}
```

For a full example, see the Secure Sockets echo client demo.

Porting

FreeRTOS Secure Sockets depends on a TCP/IP stack and on a TLS implementation. Depending on your stack, to port the Secure Sockets library, you might need to port some of the following:

- The FreeRTOS+TCP TCP/IP stack
- The corePKCS11 library
- The Transport Layer Security

For more information about porting, see Porting the Secure Sockets Library in the FreeRTOS Porting Guide.

AWS IoT Device Shadow library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

You can use the AWS IoT Device Shadow library to store and retrieve the current state (the *shadow*) of every registered device. The device's shadow is a persistent, virtual representation of your device that you can interact with in your web applications even if the device is offline. The device state is captured as its shadow in a JSON document. You can send commands to the AWS IoT Device Shadow service over MQTT or HTTP to query the latest known device state, or to change the state. Each device's shadow is uniquely identified by the name of the corresponding *thing*, a representation of a specific device or logical entity on the AWS Cloud. For more information, see Managing Devices with AWS IoT. More details about shadows can be found in AWS IoT documentation.

The AWS IoT Device Shadow library has no dependencies on additional libraries other than the standard C library. It also doesn't have any platform dependencies, such as threading or synchronization. It can be used with any MQTT library and any JSON library.

This library can be freely used and is distributed under the MIT open source license.

Code Size of AWS IoT Device Shadow (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
shadow.c	1.2K	0.9K
Total estimates	**1.2K**	**0.9K**

AWS IoT Jobs library

> **Note**
>
> The content on this page may not be up-to-date. Please refer to the FreeRTOS.org library page for the latest update.

Introduction

AWS IoT Jobs is a service that notifies one or more connected devices of a pending *job*. You can use a job to manage your fleet of devices, update firmware and security certificates on your devices, or perform administrative tasks such as restarting devices and performing diagnostics. For more information, see Jobs in the *AWS IoT Developer Guide*. Interactions with the AWS IoT Jobs service use MQTT, a lightweight publish-subscribe protocol. This library provides an API to compose and recognize the MQTT topic strings used by the AWS IoT Jobs service.

The AWS IoT Jobs library is written in C and designed to be compliant with ISO C90 and MISRA C:2012. The library has no dependencies on any additional libraries other than the standard C library. It can be used with any MQTT library and any JSON library. The library has proofs showing safe memory use and no heap allocation, making it suitable for IoT microcontrollers, but also fully portable to other platforms.

This library can be freely used and is distributed under the MIT open source license.

Code Size of AWS IoT Jobs (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
jobs.c	1.9K	1.6K

Code Size of AWS IoT Jobs (example generated with GCC for ARM Cortex-M)		
File	**With -O1 Optimization**	**With -Os Optimization**
Total estimates	**1.9K**	**1.6K**

Transport Layer Security

> **Important**
>
> This library is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The FreeRTOS Transport Layer Security (TLS) interface is a thin, optional wrapper used to abstract cryptographic implementation details away from the Secure Sockets Layer (SSL) interface above it in the protocol stack. The purpose of the TLS interface is to make the current software crypto library, mbed TLS, easy to replace with an alternative implementation for TLS protocol negotiation and cryptographic primitives. The TLS interface can be swapped out without any changes required to the SSL interface. See `iot_tls.h` in the FreeRTOS source code repository.

The TLS interface is optional because you can choose to interface directly from SSL into a crypto library. The interface is not used for MCU solutions that include a full-stack offload implementation of TLS and network transport.

For more information about porting the TLS interface, see Porting the TLS Library in the *FreeRTOS Porting Guide*.

Wi-Fi library

> **Important**
>
> This library is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Overview

The FreeRTOS Wi-Fi library abstracts port-specific Wi-Fi implementations into a common API that simplifies application development and porting for all FreeRTOS-qualified boards with Wi-Fi capabilities. Using this common API, applications can communicate with their lower-level wireless stack through a common interface.

Dependencies and requirements

The FreeRTOS Wi-Fi library requires the FreeRTOS+TCP core.

Features

The Wi-Fi library includes the following features:

- Support for WEP, WPA, WPA2, and WPA3 authentication
- Access Point Scanning
- Power management
- Network profiling

For more information about the features of the Wi-Fi library, see below.

Wi-Fi modes

Wi-Fi devices can be in one of three modes: Station, Access Point, or P2P. You can get the current mode of a Wi-Fi device by calling `WIFI_GetMode`. You can set a device's wi-fi mode by calling `WIFI_SetMode`. Switching modes by calling `WIFI_SetMode` disconnects the device, if it is already connected to a network.

Station mode

Set your device to Station mode to connect the board to an existing access point.

Access Point (AP) mode

Set your device to AP mode to make the device an access point for other devices to connect to. When your device is in AP mode, you can connect another device to your FreeRTOS device and configure the new Wi-Fi credentials. To configure AP mode, call `WIFI_ConfigureAP`. To put your device into AP mode, call `WIFI_StartAP`. To turn off AP mode, call `WIFI_StopAP`.

> **Note**
>
> FreeRTOS libraries do not provide Wi-Fi provisioning in AP mode. You must supply the additional functionality, including DHCP and HTTP server capabilities, to achieve full support of AP mode.

P2P mode

Set your device to P2P mode to allow multiple devices to connect to each other directly, without an access point.

Security

The Wi-Fi API supports WEP, WPA, WPA2, and WPA3 security types. When a device is in Station mode, you must specify the network security type when calling the `WIFI_ConnectAP` function. When a device is in AP mode, the device can be configured to use any of the supported security types:

- `eWiFiSecurityOpen`
- `eWiFiSecurityWEP`
- `eWiFiSecurityWPA`
- `eWiFiSecurityWPA2`
- `eWiFiSecurityWPA3`

Scanning and connecting

To scan for nearby access points, set your device to Station mode, and call the `WIFI_Scan` function. If you find a desired network in the scan, you can connect to the network by calling `WIFI_ConnectAP` and providing the network credentials. You can disconnect a Wi-Fi device from the network by calling `WIFI_Disconnect`. For more information about scanning and connecting, see Example usage and API reference.

Power management

Different Wi-Fi devices have different power requirements, depending on the application and available power sources. A device might always be powered on to reduce latency or it might be intermittently connected and switch into a low power mode when Wi-Fi is not required. The

interface API supports various power management modes like always on, low power, and normal mode. You set the power mode for a device using the `WIFI_SetPMMode` function. You can get the current power mode of a device by calling the `WIFI_GetPMMode` function.

Network profiles

The Wi-Fi library enables you to save network profiles in the non-volatile memory of your devices. This allows you to save network settings so they can be retrieved when a device reconnects to a Wi-Fi network, removing the need to provision devices again after they have been connected to a network. `WIFI_NetworkAdd` adds a network profile. `WIFI_NetworkGet` retrieves a network profile. `WIFI_NetworkDel` deletes a network profile. The number of profiles you can save depends on the platform.

Configuration

To use the Wi-Fi library, you need to define several identifiers in a configuration file. For information about these identifiers, see the API reference.

> **Note**
>
> The library does not include the required configuration file. You must create one. When creating your configuration file, be sure to include any board-specific configuration identifiers that your board requires.

Initialization

Before you use the Wi-Fi library, you need to initialize some board-specific components, in addition to the FreeRTOS components. Using the `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/application_code/main.c` file as a template for initialization, do the following:

1. Remove the sample Wi-Fi connection logic in `main.c` if your application handles Wi-Fi connections. Replace the following `DEMO_RUNNER_RunDemos()` function call:

```
if( SYSTEM_Init() == pdPASS )
    {
    ...
        DEMO_RUNNER_RunDemos();
    ...
        }
```

With a call to your own application:

```
if( SYSTEM_Init() == pdPASS )
    {
    ...
        // This function should create any tasks
        // that your application requires to run.
        YOUR_APP_FUNCTION();
    ...
        }
```

2. Call `WIFI_On()` to initialize and power on your Wi-Fi chip.

> **Note**
>
> Some boards might require additional hardware initialization.

3. Pass a configured `WIFINetworkParams_t` structure to `WIFI_ConnectAP()` to connect your board to an available Wi-Fi network. For more information about the `WIFINetworkParams_t` structure, see Example usage and API reference.

API reference

For a full API reference, see Wi-Fi API Reference.

Example usage

Connecting to a known AP

```
#define clientcredentialWIFI_SSID    "MyNetwork"
#define clientcredentialWIFI_PASSWORD   "hunter2"

WIFINetworkParams_t xNetworkParams;
WIFIReturnCode_t xWifiStatus;

xWifiStatus = WIFI_On(); // Turn on Wi-Fi module

// Check that Wi-Fi initialization was successful
if( xWifiStatus == eWiFiSuccess )
{
    configPRINT( ( "WiFi library initialized.\n") );
```

```
}
else
{
    configPRINT( ( "WiFi library failed to initialize.\n" ) );
    // Handle module init failure
}

/* Setup parameters. */
xNetworkParams.pcSSID = clientcredentialWIFI_SSID;
xNetworkParams.ucSSIDLength = sizeof( clientcredentialWIFI_SSID );
xNetworkParams.pcPassword = clientcredentialWIFI_PASSWORD;
xNetworkParams.ucPasswordLength = sizeof( clientcredentialWIFI_PASSWORD );
xNetworkParams.xSecurity = eWiFiSecurityWPA2;

// Connect!
xWifiStatus = WIFI_ConnectAP( &( xNetworkParams ) );

if( xWifiStatus == eWiFiSuccess )
{
    configPRINT( ( "WiFi Connected to AP.\n" ) );
    // IP Stack will receive a network-up event on success
}
else
{
    configPRINT( ( "WiFi failed to connect to AP.\n" ) );
    // Handle connection failure
}
```

Scanning for nearby APs

```
WIFINetworkParams_t xNetworkParams;
WIFIReturnCode_t xWifiStatus;

configPRINT( ("Turning on wifi...\n") );
xWifiStatus = WIFI_On();

configPRINT( ("Checking status...\n") );
if( xWifiStatus == eWiFiSuccess )
{
    configPRINT( ("WiFi module initialized.\n") );
}
else
{
```

```
    configPRINTF( ("WiFi module failed to initialize.\n" ) );
    // Handle module init failure
}

WIFI_SetMode(eWiFiModeStation);

/* Some boards might require additional initialization steps to use the Wi-Fi library.
 */

while (1)
{
    configPRINT( ("Starting scan\n") );
    const uint8_t ucNumNetworks = 12; //Get 12 scan results
    WIFIScanResult_t xScanResults[ ucNumNetworks ];
    xWifiStatus = WIFI_Scan( xScanResults, ucNumNetworks ); // Initiate scan

    configPRINT( ("Scan started\n") );

    // For each scan result, print out the SSID and RSSI
    if ( xWifiStatus == eWiFiSuccess )
    {
        configPRINT( ("Scan success\n") );
        for ( uint8_t i=0; i<ucNumNetworks; i++ )
        {
            configPRINTF( ("%s : %d \n", xScanResults[i].cSSID,
 xScanResults[i].cRSSI) );
        }
    } else {
        configPRINTF( ("Scan failed, status code: %d\n", (int)xWifiStatus) );
    }

    vTaskDelay(200);
}
```

Porting

The `iot_wifi.c` implementation needs to implement the functions defined in `iot_wifi.h`. At the very least, the implementation needs to return `eWiFiNotSupported` for any non-essential or unsupported functions.

For more information about porting the Wi-Fi library, see Porting the Wi-Fi Library in the FreeRTOS Porting Guide.

FreeRTOS demos

FreeRTOS includes some demo applications in the `demos` folder, under the main FreeRTOS directory. All of the examples that can be executed by FreeRTOS appear in the `common` folder, under `demos`. There is also a folder for each FreeRTOS-qualified platform under the `demos` folder.

Before you try the demo applications, we recommend that you complete the tutorial in Get Started with FreeRTOS. It shows you how to set up and run the coreMQTT Agent demo.

Running the FreeRTOS demos

The following topics show you how to set up and run the FreeRTOS demos:

- Bluetooth Low Energy demo applications
- Demo bootloader for the Microchip Curiosity PIC32MZEF
- AWS IoT Device Defender demo
- AWS IoT Greengrass V1 discovery demo application
- AWS IoT Greengrass V2
- coreHTTP demos
- AWS IoT Jobs library demo
- coreMQTT demos
- Over-the-air updates demo application
- Secure Sockets echo client demo
- AWS IoT Device Shadow demo application

The `DEMO_RUNNER_RunDemos` function, located in the *`freertos`*`/demos/demo_runner/iot_demo_runner.c` file, initializes a detached thread on which a single demo application runs. By default, `DEMO_RUNNER_RunDemos` only calls and starts the coreMQTT Agent demo. Depending on the configuration that you selected when you downloaded FreeRTOS, and where you downloaded FreeRTOS, the other example runner functions might start by default. To enable a demo application, open the *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h` file, and define the demo that you want to run.

Note

Not all combinations of examples work together. Depending on the combination, the software might not run on the selected target due to memory constraints. We recommend that you run one demo at a time.

Configuring the demos

The demos have been configured to get you started quickly. You might want to change some of the configurations for your project to create a version that runs on your platform. You can find configuration files at `vendors/vendor/boards/board/aws_demos/config_files`.

Bluetooth Low Energy demo applications

Important

This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Overview

FreeRTOS Bluetooth Low Energy includes three demo applications:

- MQTT over Bluetooth Low Energy demo

  This application demonstrates how to use the MQTT over Bluetooth Low Energy service.

- Wi-Fi provisioning demo

  This application demonstrates how to use the Bluetooth Low Energy Wi-Fi Provisioning service.

- Generic Attributes Server demo

  This application demonstrates how to use the FreeRTOS Bluetooth Low Energy middleware APIs to create a simple GATT server.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Prerequisites

To follow along with these demos, you need a microcontroller with Bluetooth Low Energy capabilities. You also need the iOS SDK for FreeRTOS Bluetooth devices or the Android SDK for FreeRTOS Bluetooth devices.

Set up AWS IoT and Amazon Cognito for FreeRTOS Bluetooth Low Energy

To connect your devices to AWS IoT across MQTT, you need to set up AWS IoT and Amazon Cognito.

To set up AWS IoT

1. Set up an AWS account on https://aws.amazon.com/.
2. Open the AWS IoT console, and from the navigation pane, choose **Manage**, and then choose **Things**.
3. Choose **Create**, and then choose **Create a single thing**.
4. Enter a name for your device, and then choose **Next**.
5. If you are connecting your microcontroller to the cloud through a mobile device, choose **Create thing without certificate**. Because the Mobile SDKs use Amazon Cognito for device authentication, you do not need to create a device certificate for demos that use Bluetooth Low Energy.

   If you are connecting your microcontroller to the cloud directly over Wi-Fi, choose **Create certificate**, choose **Activate**, and then download the thing's certificate, public key, and private key.
6. Choose the thing that you just created from the list of registered things, and then choose **Interact** from your thing's page. Make a note of the AWS IoT REST API endpoint.

For more information about setting up, see the Getting Started with AWS IoT.

To create an Amazon Cognito user pool

1. Open the Amazon Cognito console, and choose **Manage User Pools**.
2. Choose **Create a user pool**.
3. Give the user pool a name, and then choose **Review defaults**.
4. From the navigation pane, choose **App clients**, and then choose **Add an app client**.
5. Enter a name for the app client, and then choose **Create app client**.
6. From the navigation pane, choose **Review**, and then choose **Create pool**.

   Make a note of the pool ID that appears on the **General Settings** page of your user pool.
7. From the navigation pane, choose **App clients**, and then choose **Show details**. Make a note of the app client ID and app client secret.

To create an Amazon Cognito identity pool

1. Open the Amazon Cognito console, and choose **Manage Identity Pools**.
2. Enter a name for your identity pool.
3. Expand **Authentication providers**, choose the **Cognito** tab, and then enter your user pool ID and app client ID.
4. Choose **Create Pool**.
5. Expand **View Details**, and make a note of the two IAM role names. Choose **Allow** to create the IAM roles for authenticated and unauthenticated identities to access Amazon Cognito.
6. Choose **Edit identity pool**. Make a note of the identity pool ID. It should be of the form `us-west-2:12345678-1234-1234-1234-123456789012`.

For more information about setting up Amazon Cognito, see the Getting Started with Amazon Cognito.

To create and attach an IAM policy to the authenticated identity

1. Open the IAM console, and from the navigation pane, choose **Roles**.
2. Find and choose your authenticated identity's role, choose **Attach policies**, and then choose **Add inline policy**.
3. Choose the **JSON** tab, and paste the following JSON:

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "iot:AttachPolicy",
            "iot:AttachPrincipalPolicy",
            "iot:Connect",
            "iot:Publish",
            "iot:Subscribe",
            "iot:Receive",
            "iot:GetThingShadow",
            "iot:UpdateThingShadow",
            "iot:DeleteThingShadow"
         ],
         "Resource":[
            "*"
         ]
      }
   ]
}
```

4. Choose **Review policy**, enter a name for the policy, and then choose **Create policy**.

Keep your AWS IoT and Amazon Cognito information on hand. You need the endpoint and IDs to authenticate your mobile application with the AWS Cloud.

Set up your FreeRTOS environment for Bluetooth Low Energy

To set up your environment, you need to download FreeRTOS with the Bluetooth Low Energy library on your microcontroller, and download and configure the Mobile SDK for FreeRTOS Bluetooth Devices on your mobile device.

To set up your microcontroller's environment with FreeRTOS Bluetooth Low Energy

1. Download or clone FreeRTOS from GitHub. See the README.md file for instructions.
2. Set up FreeRTOS on your microcontroller.

For information about getting started with FreeRTOS on a FreeRTOS-qualified microcontroller, see the guide for your board in Getting Started with FreeRTOS.

> **Note**
>
> You can run the demos on any Bluetooth Low Energy-enabled microcontroller with FreeRTOS and ported FreeRTOS Bluetooth Low Energy libraries. Currently, the FreeRTOS MQTT over Bluetooth Low Energy demo project is fully ported to the following Bluetooth Low Energy-enabled devices:
>
> - Espressif ESP32-DevKitC and the ESP-WROVER-KIT
> - Nordic nRF52840-DK

Common components

The FreeRTOS demo applications have two common components:

- Network Manager
- Bluetooth Low Energy Mobile SDK demo application

Network Manager

Network Manager manages your microcontroller's network connection. It is located in your FreeRTOS directory at `demos/network_manager/aws_iot_network_manager.c`. If the Network Manager is enabled for both Wi-Fi and Bluetooth Low Energy, the demos start with Bluetooth Low Energy by default. If the Bluetooth Low Energy connection is disrupted, and your board is Wi-Fi-enabled, the Network Manager switches to an available Wi-Fi connection to prevent you from disconnecting from the network.

To enable a network connection type with the Network Manager, add the network connection type to the `configENABLED_NETWORKS` parameter in `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_iot_network_config.h` (where the *`vendor`* is the name of the vendor and the *`board`* is the name of the board that you are using to run the demos).

For example, if you have both Bluetooth Low Energy and Wi-Fi enabled, the line that starts with `#define configENABLED_NETWORKS` in `aws_iot_network_config.h` reads as follows:

```
#define  configENABLED_NETWORKS  ( AWSIOT_NETWORK_TYPE_BLE | AWSIOT_NETWORK_TYPE_WIFI )
```

To get a list of currently supported network connection types, see the lines that begin with `#define AWSIOT_NETWORK_TYPE` in `aws_iot_network.h`.

FreeRTOS Bluetooth Low Energy Mobile SDK demo application

The FreeRTOS Bluetooth Low Energy Mobile SDK demo application is located on GitHub at Android SDK for FreeRTOS Bluetooth Devices under `amazon-freertos-ble-android-sdk/app` and the iOS SDK for FreeRTOS Bluetooth Devices under `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo`. In this example, we use screenshots of the iOS version of the demo mobile application.

> **Note**
>
> If you are using an iOS device, you need Xcode to build the demo mobile application. If you are using an Android device, you can use Android Studio to build the demo mobile application.

To configure the iOS SDK demo application

When you define configuration variables, use the format of the placeholder values provided in the configuration files.

1. Confirm that the iOS SDK for FreeRTOS Bluetooth devices is installed.
2. Issue the following command from `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/`:

   ```
   $ pod install
   ```

3. Open the `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo.xcworkspace` project with Xcode, and change the signing developer account to your account.
4. Create an AWS IoT policy in your region (if you haven't already).

> **Note**
>
> This policy is different from the IAM policy created for the Amazon Cognito authenticated identity.

a. Open the AWS IoT console.

b. In the navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**. Enter a name to identify your policy. In the **Add statements** section, choose **Advanced mode**. Copy and paste the following JSON into the policy editor window. Replace *aws-region* and *aws-account* with your AWS Region and account ID.

c. Choose **Create**.

5. Open `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo/Amazon/AmazonConstants.swift`, and redefine the following variables:

   - `region`: Your AWS Region.
   - `iotPolicyName`: Your AWS IoT policy name.
   - `mqttCustomTopic`: The MQTT topic that you want to publish to.

6. Open `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo/Support/awsconfiguration.json`.

   Under `CognitoIdentity`, redefine the following variables:

   - `PoolId`: Your Amazon Cognito identity pool ID.
   - `Region`: Your AWS Region.

   Under `CognitoUserPool`, redefine the following variables:

   - `PoolId`: Your Amazon Cognito user pool ID.
   - `AppClientId`: Your app client ID.
   - `AppClientSecret`: Your app client secret.
   - `Region`: Your AWS Region.

To configure the Android SDK demo application

When you define configuration variables, use the format of the placeholder values provided in the configuration files.

1. Confirm that the Android SDK for FreeRTOS Bluetooth devices is installed.
2. Create an AWS IoT policy in your region (if you haven't already).

   > **Note**
   >
   > This policy is different from the IAM policy created for the Amazon Cognito authenticated identity.

   a. Open the AWS IoT console.

   b. In the navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**. Enter a name to identify your policy. In the **Add statements** section, choose **Advanced mode**. Copy and paste the following JSON into the policy editor window. Replace *aws-region* and *aws-account* with your AWS Region and account ID.

   c. Choose **Create**.
3. Open https://github.com/aws/amazon-freertos-ble-android-sdk/blob/master/app/src/main/java/software/amazon/freertos/demo/DemoConstants.java and redefine the following variables:

   - `AWS_IOT_POLICY_NAME`: Your AWS IoT policy name.
   - `AWS_IOT_REGION`: Your AWS Region.
4. Open https://github.com/aws/amazon-freertos-ble-android-sdk/blob/master/app/src/main/res/raw/awsconfiguration.json.

   Under `CognitoIdentity`, redefine the following variables:

   - `PoolId`: Your Amazon Cognito identity pool ID.
   - `Region`: Your AWS Region.

   Under `CognitoUserPool`, redefine the following variables:

   - `PoolId`: Your Amazon Cognito user pool ID.

- `AppClientId`: Your app client ID.
- `AppClientSecret`: Your app client secret.
- `Region`: Your AWS Region.

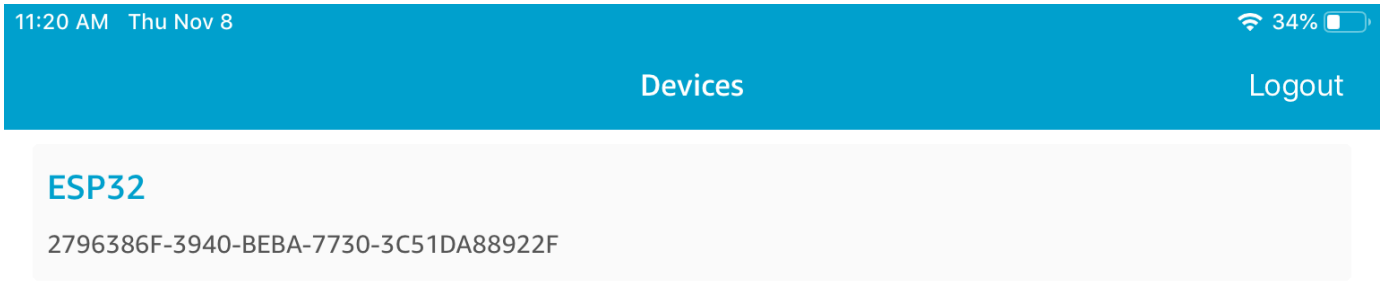
To discover and establish secure connections with your microcontroller over Bluetooth Low Energy

1. In order to pair your microcontroller and mobile device securely (step 6), you need a serial terminal emulator with both input and output capabilities (such as TeraTerm). Configure the terminal to connect to your board by a serial connection as instructed in Installing a terminal emulator.
2. Run the Bluetooth Low Energy demo project on your microcontroller.
3. Run the Bluetooth Low Energy Mobile SDK demo application on your mobile device.

   To start the demo application in the Android SDK from the command line, run the following command:

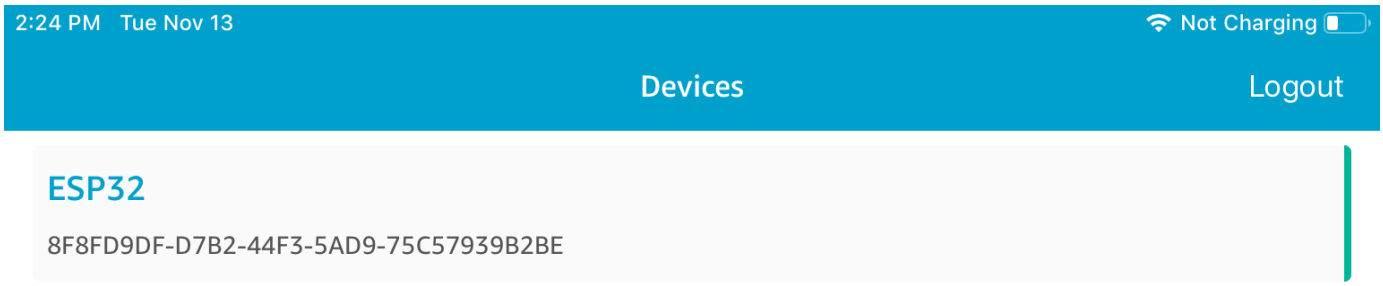
   ```
   $ ./gradlew installDebug
   ```

4. Confirm that your microcontroller appears under **Devices** on the Bluetooth Low Energy Mobile SDK demo app.

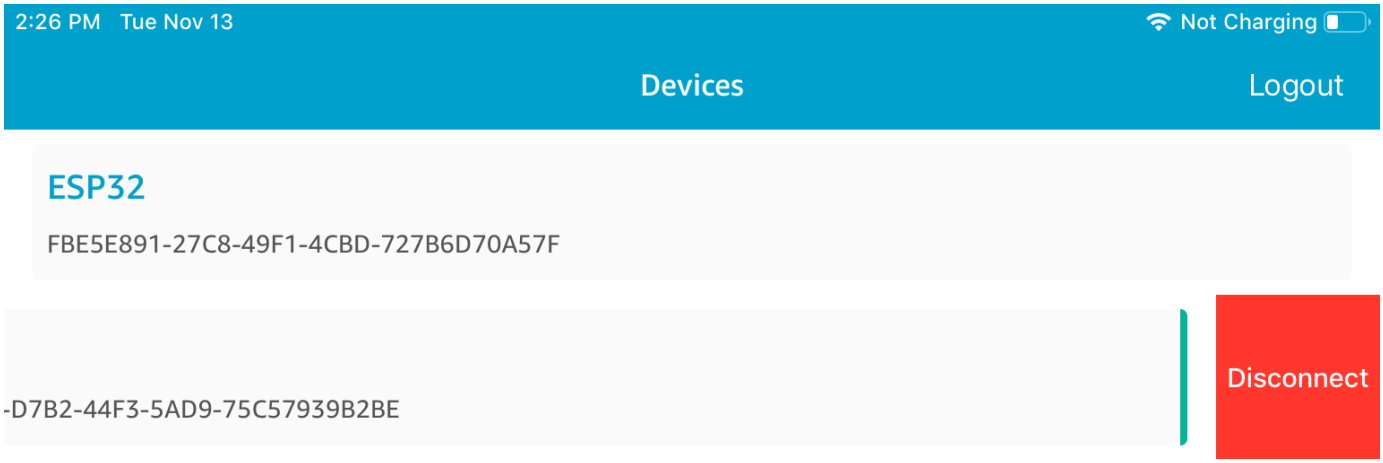


> **Note**
>
> All devices with FreeRTOS and the device information service (*freertos*`/.../device_information`) that are in range appear in the list.

5. Choose your microcontroller from the list of devices. The application establishes a connection with the board, and a green line appears next to the connected device.

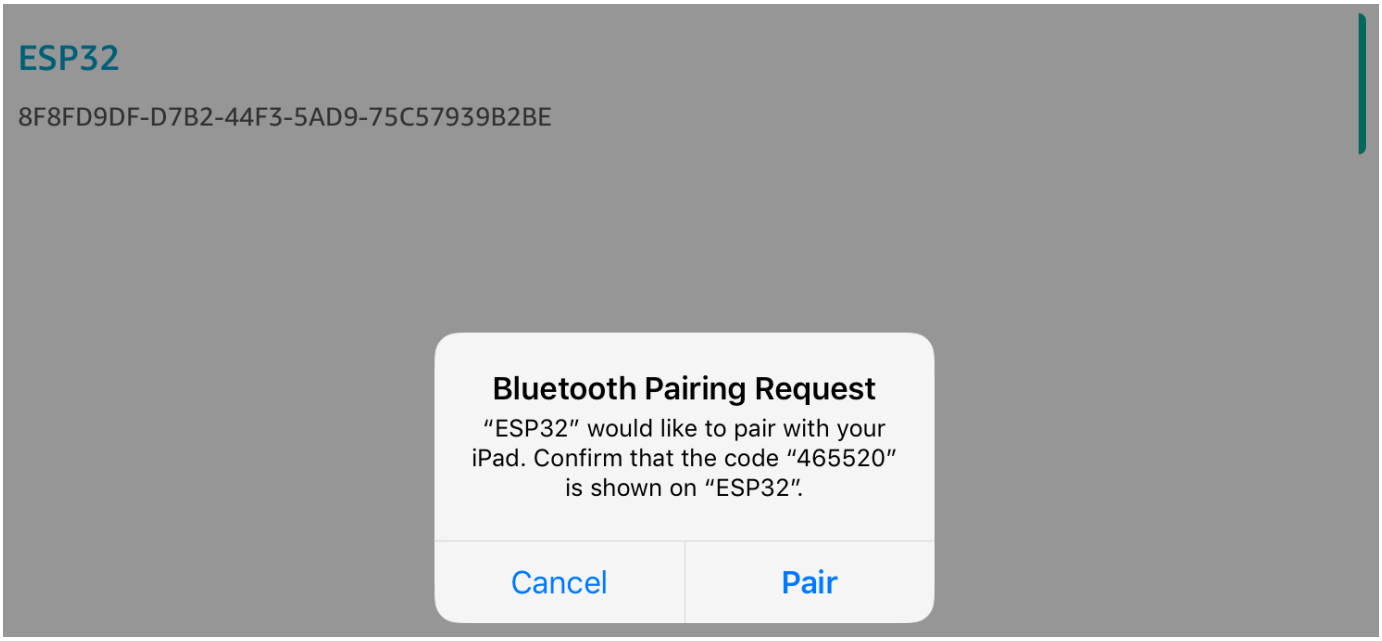


You can disconnect from your microcontroller by dragging the line to the left.



6. If prompted, pair your microcontroller and mobile device.

```
24 261107 [Btc_task]  Disconnected from BLE device. Stopping the counter update
25 261108 [Btc_task] Disconnect received for MQTT service instance 0
26 261108 [Btc_task] BLE disconnected with remote device, start advertisement
27 261108 [Btc_task] Started advertisement. Listening for a BLE Connection.
28 261289 [Btc_task] BLE Connected to remote device, connId = 0
E (2614110) BT_GATT: gatts_write_attr_perm_check - GATT_INSUF_AUTHENTICATION: MITM required
E (2614420) BT_SMP: Value for numeric comparison = 465520
29 261412 [uTask] Numeric comparison:465520
30 261412 [uTask] Press 'y' to confirm
```



If the code for numeric comparison is the same on both devices, pair the devices.

> **Note**
>
> The Bluetooth Low Energy Mobile SDK demo application uses Amazon Cognito for user authentication. Make sure that you have set up a Amazon Cognito user and identity pools, and that you have attached IAM policies to authenticated identities.

MQTT over Bluetooth Low Energy

In the MQTT over Bluetooth Low Energy demo, your microcontroller publishes messages to the AWS Cloud through an MQTT proxy.

To subscribe to a demo MQTT topic

1. Sign in to the AWS IoT console.
2. In the navigation pane, choose **Test**, then choose **MQTT test client** to open the MQTT client.
3. In **Subscription topic**, enter ***thing-name*`/example/topic1`**, and then choose **Subscribe to topic**.

If you use Bluetooth Low Energy to pair the microcontroller with your mobile device, the MQTT messages are routed through the Bluetooth Low Energy Mobile SDK demo application on your mobile device.

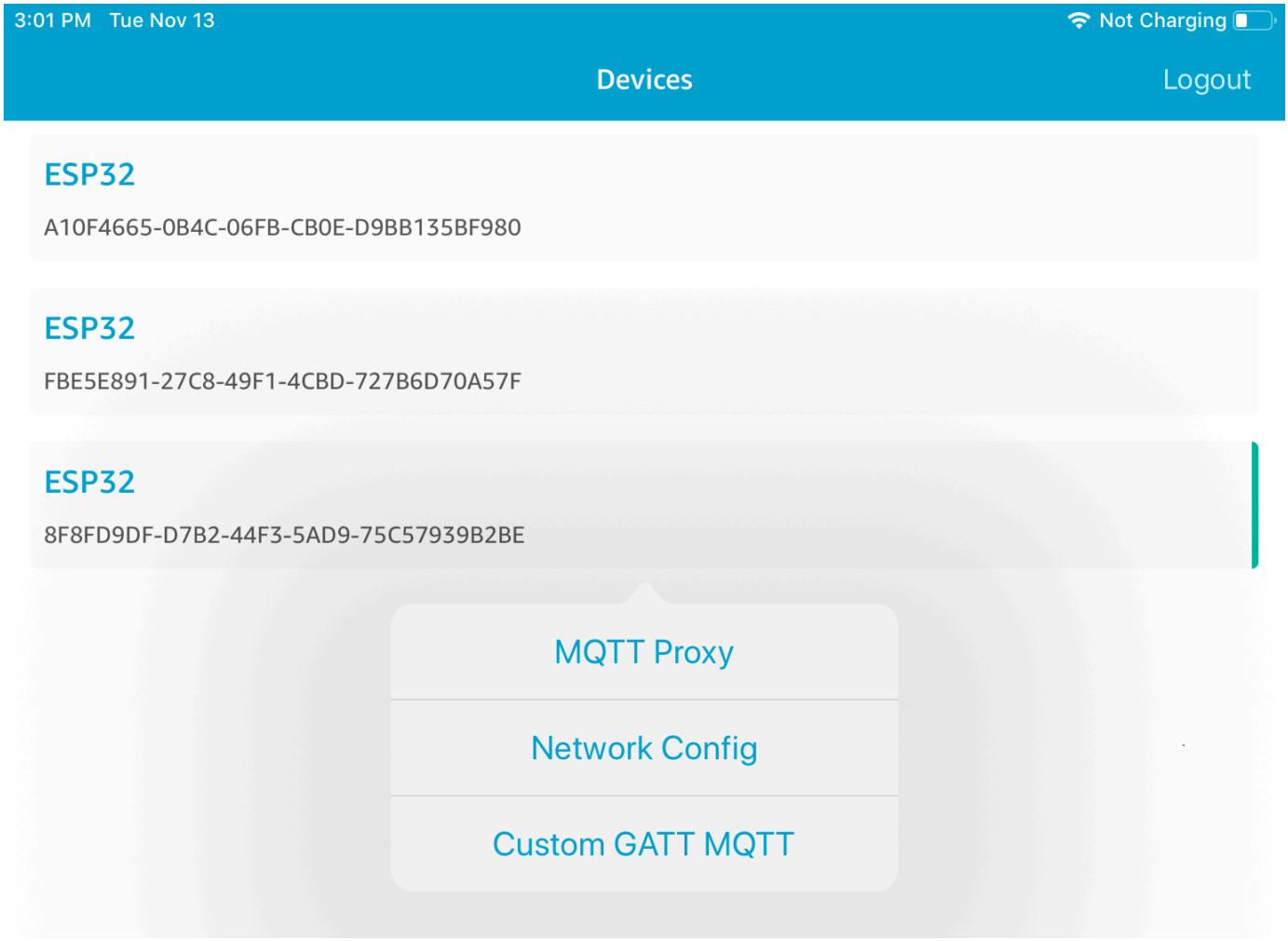
To enable the demo over Bluetooth Low Energy

1. Open `vendors/vendor/boards/board/aws_demos/config_files/aws_demo_config.h`, and define `CONFIG_MQTT_BLE_TRANSPORT_DEMO_ENABLED`.
2. Open `demos/include/aws_clientcredential.h`, and configure `clientcredentialMQTT_BROKER_ENDPOINT` with the AWS IoT broker endpoint. Configure `clientcredentialIOT_THING_NAME` with the thing name for the BLE micro controller device. The AWS IoT broker endpoint can be obtained from the AWS IoT console by choosing **Settings** in the left navigation pane, or through the CLI by running the command: `aws iot describe-endpoint --endpoint-type=iot:Data-ATS`.

   > **Note**
   >
   > The AWS IoT broker endpoint and thing name must both be in the same region where the cognito identity and user pool are configured.

To run the demo

1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your board and your mobile device using the FreeRTOS Bluetooth Low Energy Mobile SDK demo application.
3. From the **Devices** list in the demo mobile app, choose your microcontroller, and then choose **MQTT Proxy** to open the MQTT proxy settings.



4. After you enable the MQTT proxy, MQTT messages appear on the *thing-name*`/example/topic1` topic, and data is printed to the UART terminal.

Wi-Fi provisioning

Wi-Fi Provisioning is a FreeRTOS Bluetooth Low Energy service that allows you to securely send Wi-Fi network credentials from a mobile device to a microcontroller over Bluetooth Low Energy. The source code for the Wi-Fi Provisioning service can be found at *freertos*`/.../wifi_provisioning`.

> **Note**
>
> The Wi-Fi Provisioning demo is currently supported on the Espressif ESP32-DevKitC.

To enable the demo

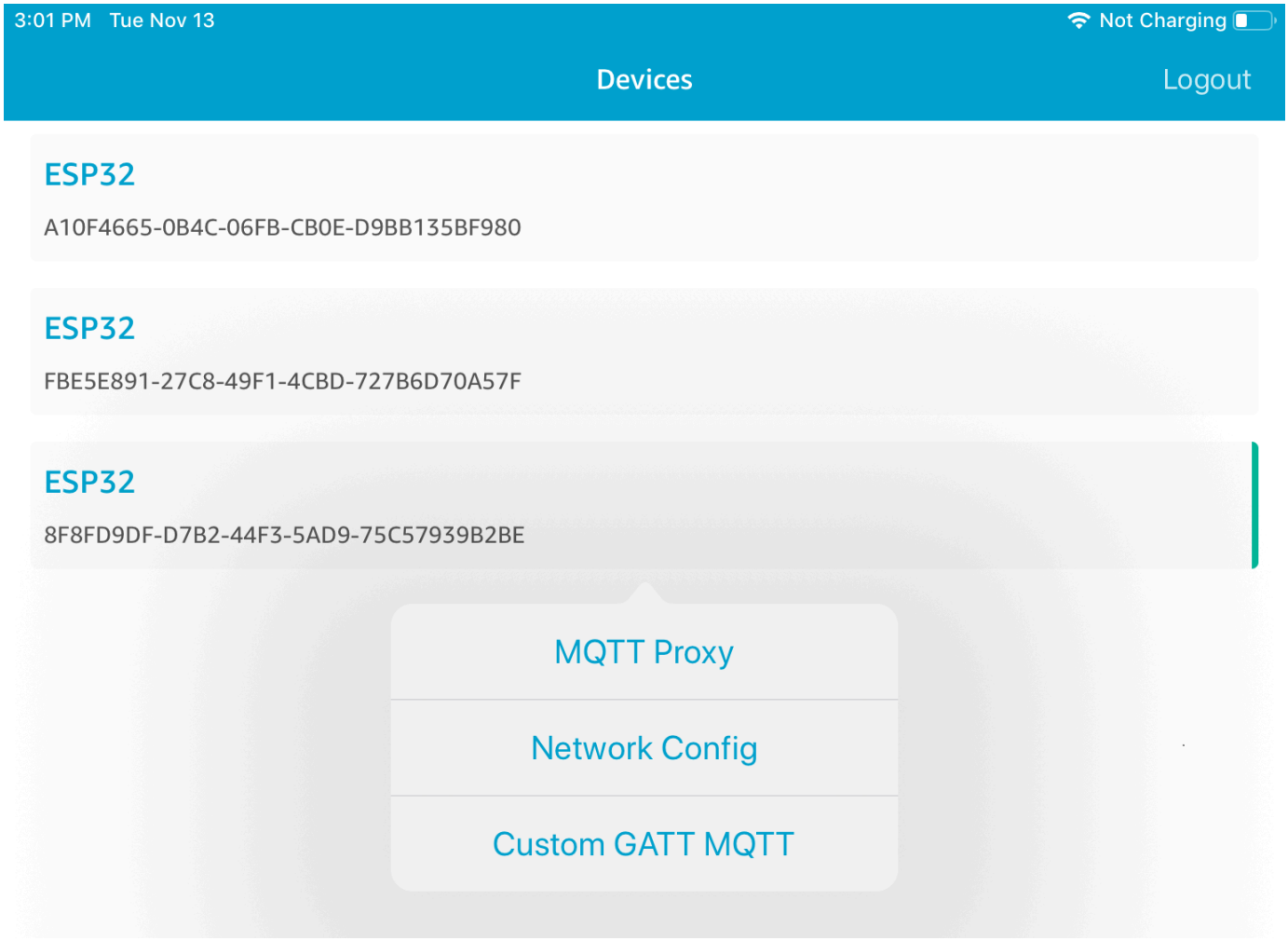
1. Enable the Wi-Fi Provisioning service. Open `vendors/vendor/boards/board/aws_demos/config_files/iot_ble_config.h`, and set `#define IOT_BLE_ENABLE_WIFI_PROVISIONING` to `1` (where the *vendor* is the name of the vendor and the *board* is the name of the board that you are using to run the demos).

   > **Note**
   >
   > The Wi-Fi Provisioning service is disabled by default.

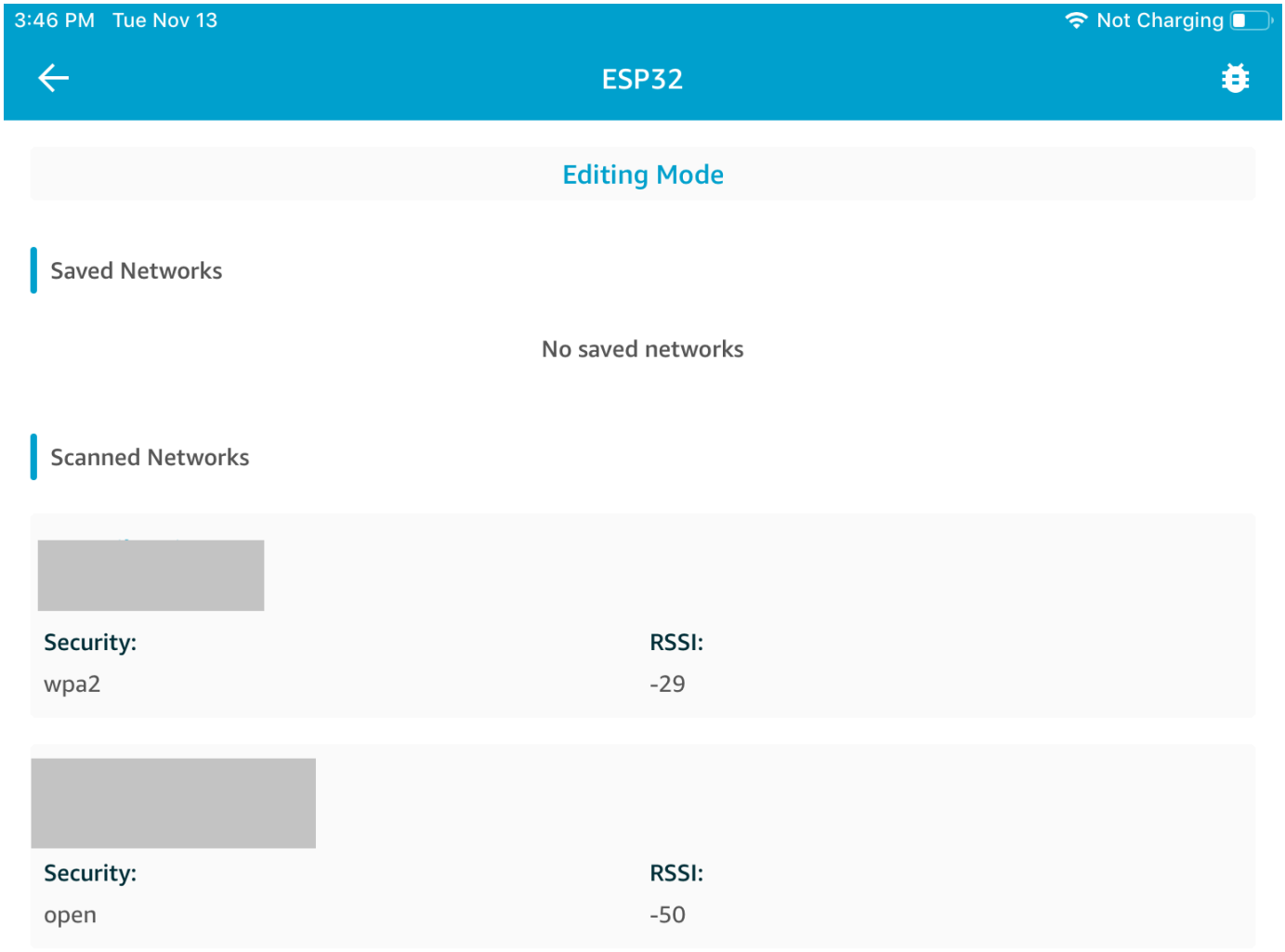
2. Configure the Network Manager to enable both Bluetooth Low Energy and Wi-Fi.

To run the demo

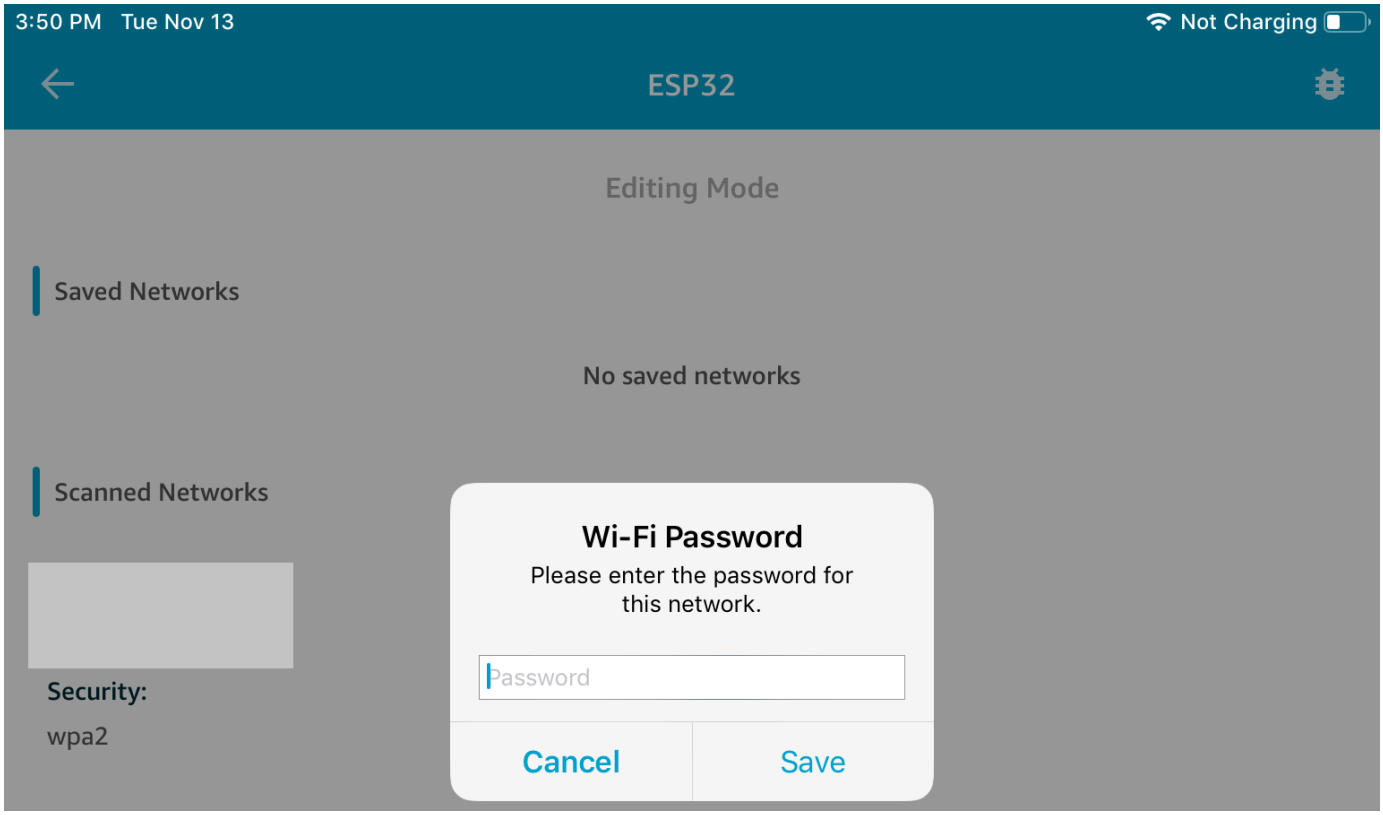
1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your microcontroller and your mobile device using the FreeRTOS Bluetooth Low Energy Mobile SDK demo application.
3. From the **Devices** list in the demo mobile app, choose your microcontroller, and then choose **Network Config** to open the network configuration settings.



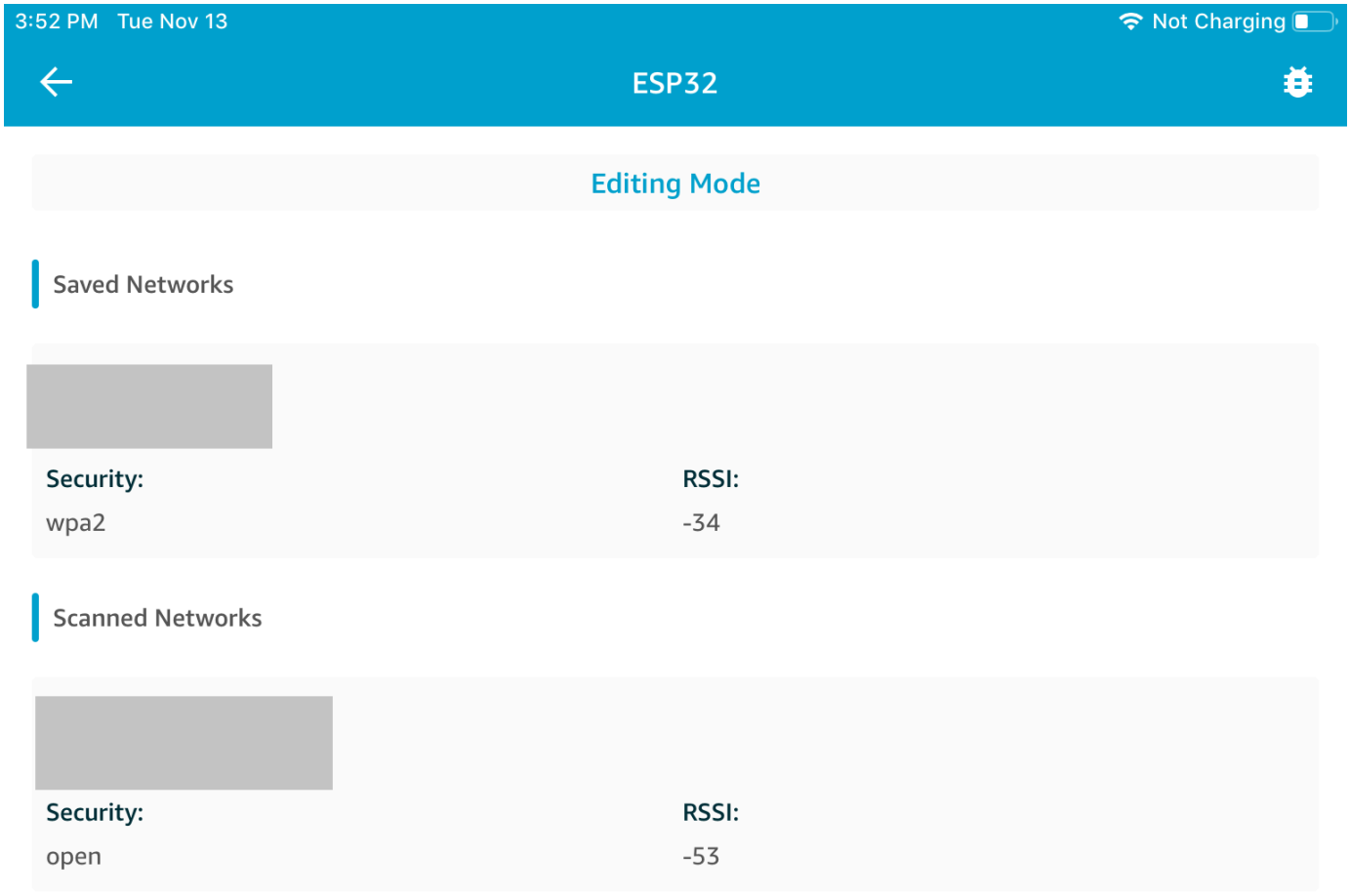
4. After you choose **Network Config** for your board, the microcontroller sends a list of the networks in the vicinity to the mobile device. Available Wi-Fi networks appear in a list under **Scanned Networks**.



From the **Scanned Networks** list, choose your network, and then enter the SSID and password, if required.

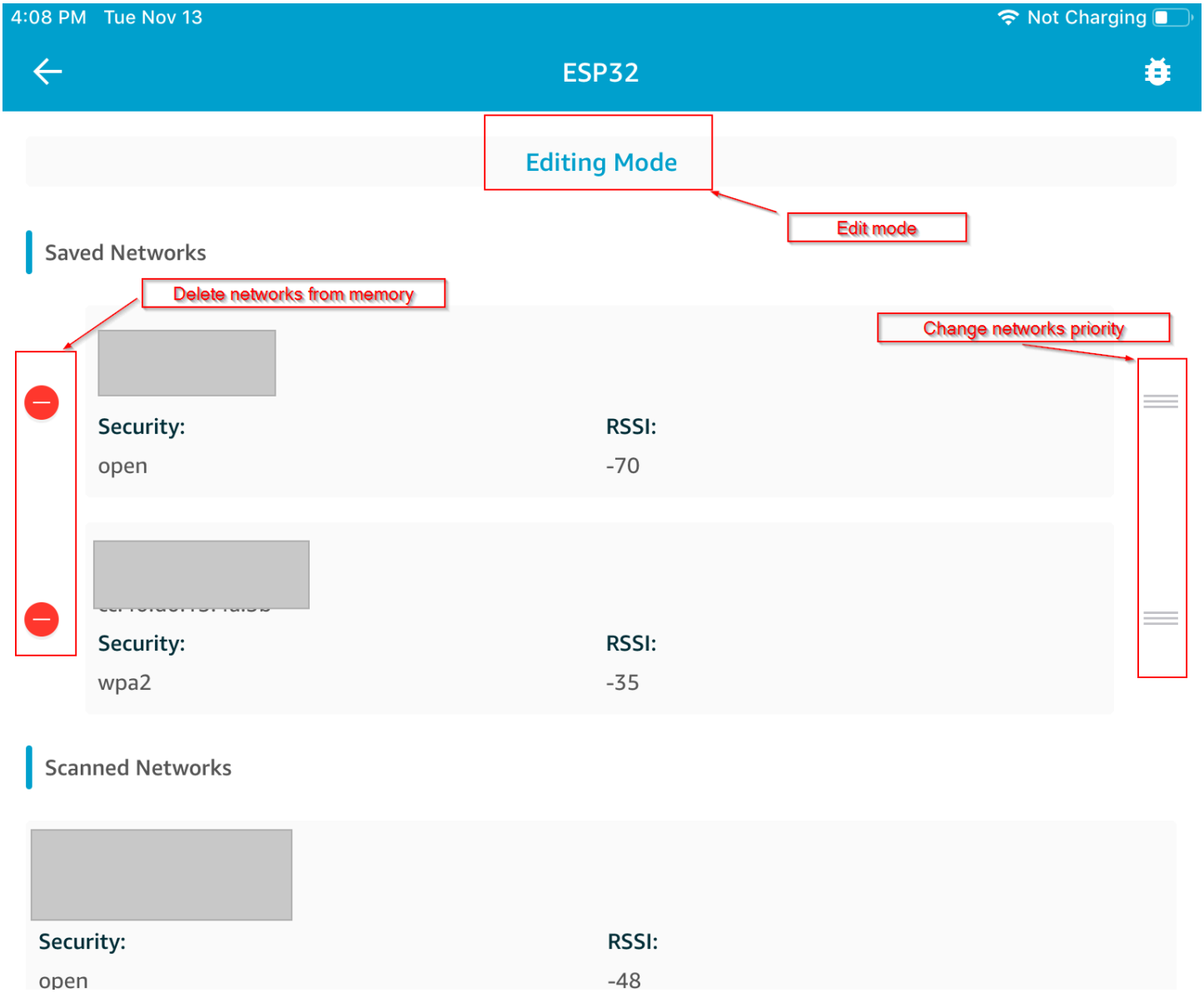


The microcontroller connects to and saves the network. The network appears under **Saved Networks**.



You can save several networks in the demo mobile app. When you restart the application and demo, the microcontroller connects to the first available saved network, starting from the top of the **Saved Networks** list.

To change the network priority order or delete networks, on the **Network Configuration** page, choose **Editing Mode**. To change the network priority order, choose the right side of the network that you want to reprioritize, and drag the network up or down. To delete a network, choose the red button on the left side of the network that you want to delete.



Generic Attributes Server

In this example, a demo Generic Attributes (GATT) Server application on your microcontroller sends a simple counter value to the FreeRTOS Bluetooth Low Energy Mobile SDK demo application.

Using the Bluetooth Low Energy Mobile SDKs, you can create your own GATT client for a mobile device that connects to the GATT server on your microcontroller and runs in parallel with the demo mobile application.

To enable the demo

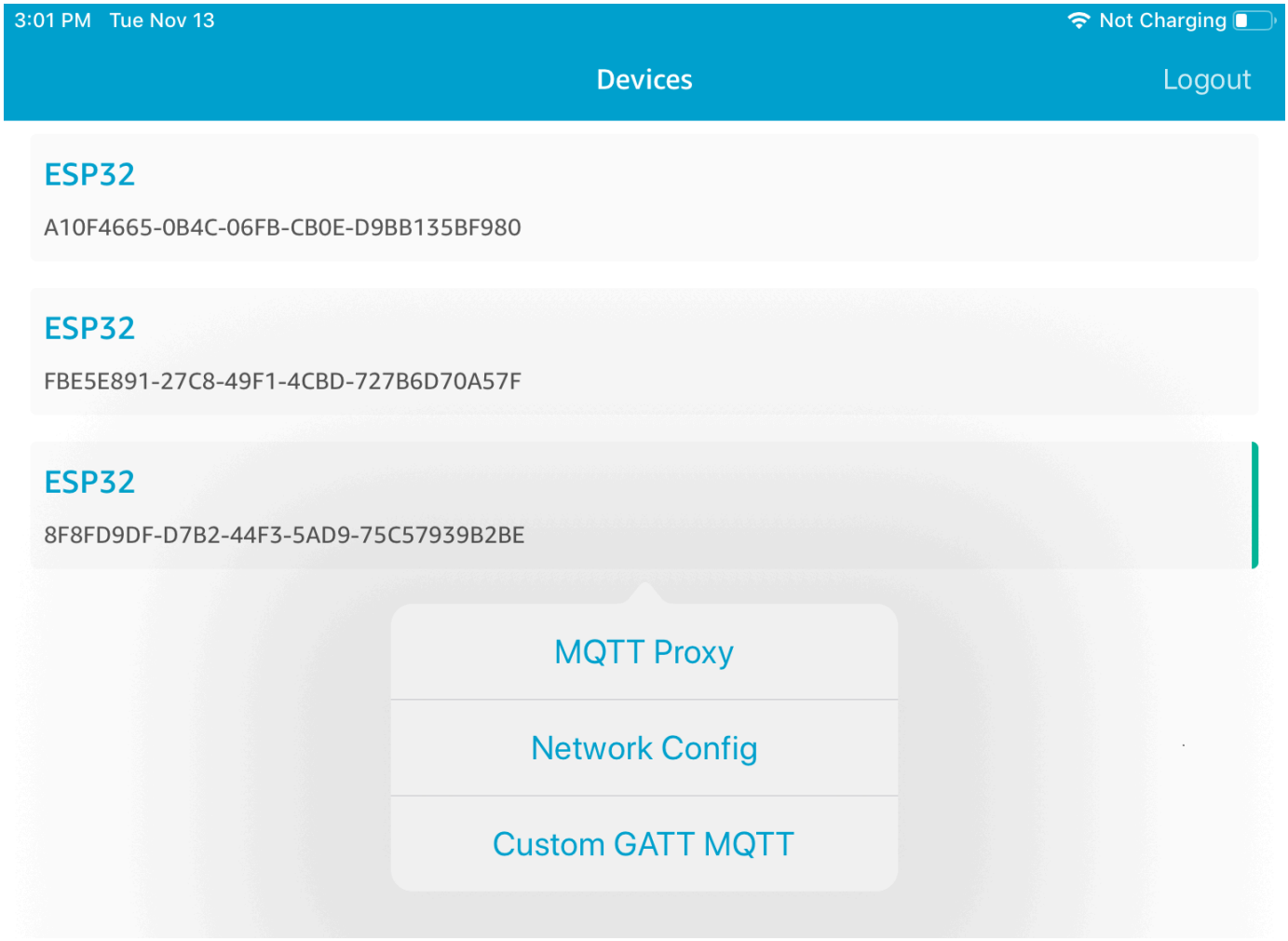
1. Enable the Bluetooth Low Energy GATT demo. In `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/iot_ble_config.h` (where the *`vendor`* is the name of the vendor and the *`board`* is the name of the board that you are using to run the demos), add `#define IOT_BLE_ADD_CUSTOM_SERVICES ( 1 )` to the list of define statements.

   > **Note**
   >
   > The Bluetooth Low Energy GATT demo is disabled by default.

2. Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED`, and define `CONFIG_BLE_GATT_SERVER_DEMO_ENABLED`.

To run the demo

1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your board and your mobile device using the FreeRTOS Bluetooth Low Energy Mobile SDK demo application.
3. From the **Devices** list in the app, choose your board, and then choose **MQTT Proxy** to open the MQTT proxy options.



4. Return to the **Devices** list, choose your board, and then choose **Custom GATT MQTT** to open the custom GATT service options.
5. Choose **Start Counter** to start publishing data to the ***your-thing-name*/example/topic** MQTT topic.

   After you enable the MQTT proxy, Hello World and incrementing counter messages appear on the *your-thing-name*/example/topic topic.

Demo bootloader for the Microchip Curiosity PIC32MZEF

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an

existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Note

In agreement with Microchip, we are removing the Curiosity PIC32MZEF (DM320104) from the FreeRTOS Reference Integration repository main branch and will no longer carry it in new releases. Microchip has issued an official notice that the PIC32MZEF (DM320104) is no longer recommended for new designs. The PIC32MZEF projects and source code can still be accessed through the previous release tags. Microchip recommends that customers use the Curiosity PIC32MZ-EF-2.0 Development board (DM320209) for new designs. The PIC32MZv1 platform can still be found in v202012.00 of the FreeRTOS Reference Integration repository. However, the platform is no longer supported by v202107.00 of the FreeRTOS Reference.

This demo bootloader implements firmware version checking, cryptographic signature verification, and application self-testing. These capabilities support over-the-air (OTA) firmware updates for FreeRTOS.

The firmware verification includes verifying the authenticity and integrity of the new firmware received over the air. The bootloader verifies the cryptographic signature of the application before booting. The demo uses elliptic-curve digital signature algorithm (ECDSA) over SHA-256. The utilities provided can be used to generate a signed application that can be flashed on the device.

The bootloader supports the following features required for OTA:

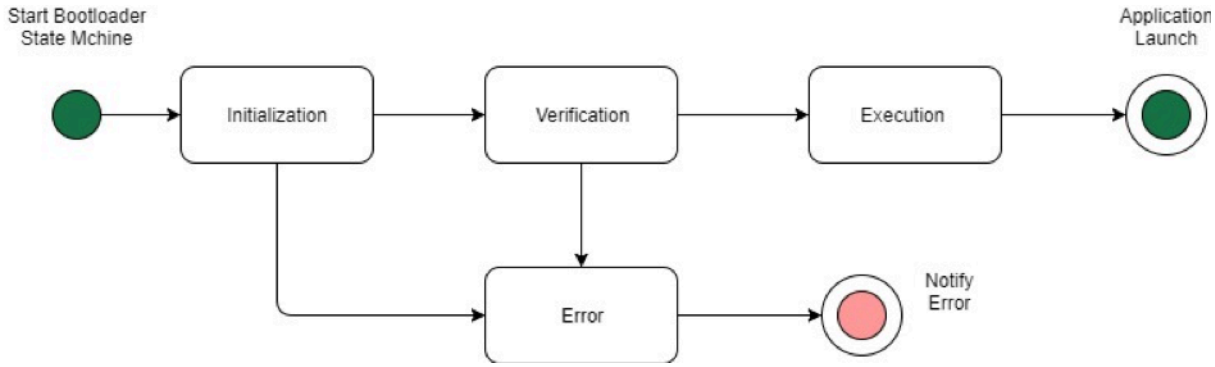
- Maintains application images on the device and switches between them.
- Allows self-test execution of a received OTA image and rollback on failure.
- Checks signature and version of the OTA update image.

Note

To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Bootloader states

The bootloader process is shown in the following state machine.



The following table describes the bootloader states.

Bootloader State	Description
Initialization	Bootloader is in the initialization state.
Verification	Bootloader is verifying the images present on the device.
Execute Image	Bootloader is launching the selected image.
Execute Default	Bootloader is launching the default image.
Error	Bootloader is in the error state.

In the preceding diagram, both `Execute Image` and `Execute Default` are shown as the `Execution` state.

Bootloader Execution State

The bootloader is in the `Execution` state and is ready to launch the selected verified image. If the image to be launched is in the upper bank, the banks are swapped before executing the image, because the application is always built for the lower bank.

Bootloader Default Execution State

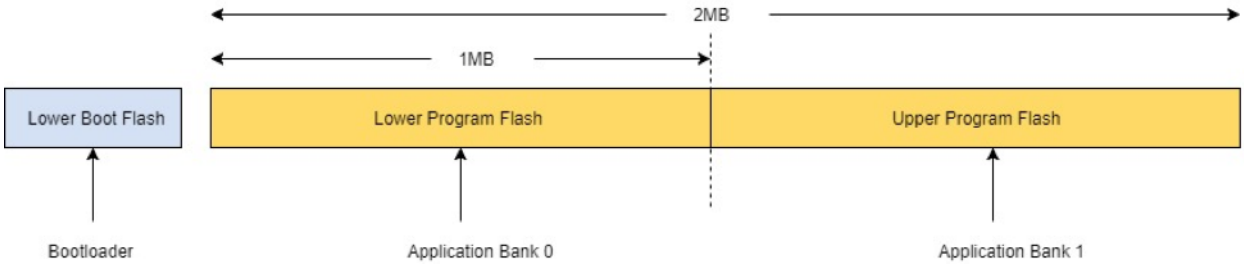
If the configuration option to launch the default image is enabled, the bootloader launches the application from a default execution address. This option must be disabled except while debugging.

Bootloader Error State

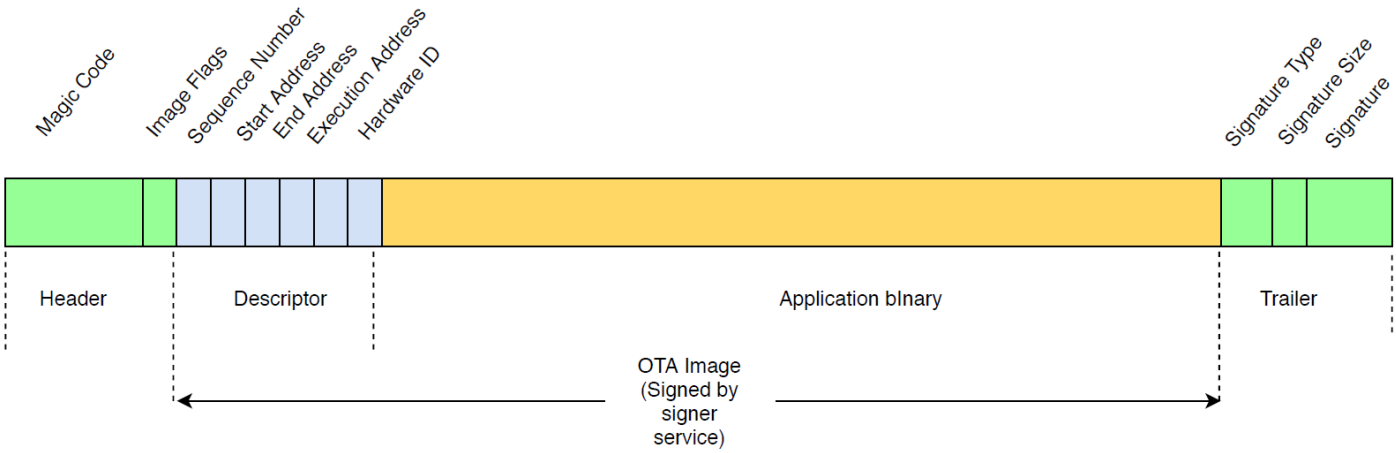
The bootloader is in an error state and no valid images are present on the device. The bootloader must notify the user. The default implementation sends a log message to the console and fast-blinks the LED on the board indefinitely.

Flash device

The Microchip Curiosity PIC32MZEF platform contains an internal program flash of two megabytes (MB) divided into two banks. It supports memory map swapping between these two banks and live updates. The demo bootloader is programmed in a separate lower boot flash region.



Application image structure



The diagram shows the primary components of the application image stored on each bank of the device.

Component	Size (in bytes)
Image header	8 bytes
Image descriptor	24 bytes
Application binary	< 1 MB - (324)
Trailer	292 bytes

Image header

The application images on the device must start with a header that consists of a magic code and image flags.

Header Field	Size (in bytes)
Magic code	7 bytes
Image flags	1 byte

Magic code

The image on the flash device must start with a magic code. The default magic code is `@AFRTOS`. The bootloader checks if a valid magic code is present before booting the image. This is the first step of verification.

Image flags

The image flags are used to store the status of the application images. The flags are used in the OTA process. The image flags of both banks determine the state of the device. If the executing image is marked as commit pending, it means the device is in the OTA self-test phase. Even if images on the devices are marked valid, they go through the same verification steps on every boot. If an image is marked as new, the bootloader marks it as commit pending and launches it for self-test after verification. The bootloader also initializes and starts the watchdog timer so that if the new OTA image fails self-test, the device reboots and bootloader rejects the image by erasing it and executes the previous valid image.

The device can have only one valid image. The other image can be a new OTA image or a commit pending (self-test). After a successful OTA update, the old image is erased from the device.

Status	Value	Description
New image	0xFF	Application image is new and never executed.
Commit pending	0xFE	Application image is marked for test execution.
Valid	0xFC	Application image is marked valid and committed.
Invalid	0xF8	Application image is marked invalid.

Image descriptor

The application image on the flash device must contain the image descriptor following the image header. The image descriptor is generated by a post-build utility that uses configuration files (`ota-descriptor.config`) to generate the appropriate descriptor and prepends it to the application binary. The output of this post-build step is the binary image that can be used for OTA.

Descriptor Field	Size (in bytes)
Sequence Number	4 bytes
Start Address	4 bytes
End Address	4 bytes
Execution Address	4 bytes
Hardware ID	4 bytes
Reserved	4 bytes

Sequence Number

The sequence number must be incremented before building a new OTA image. See the `ota-descriptor.config` file. The bootloader uses this number to determine the image to boot. Valid values are from 1 to 4294967295.

Start Address

The starting address of the application image on the device. As the image descriptor is prepended to the application binary, this address is the start of the image descriptor.

End Address

The ending address of the application image on the device, excluding the image trailer.

Execution Address

The execution address of the image.

Hardware ID

A unique hardware ID used by the bootloader to verity the OTA image is built for the correct platform.

Reserved

This is reserved for future use.

Image trailer

The image trailer is appended to the application binary. It contains the signature type string, signature size, and signature of the image.

Trailer Field	Size (in bytes)
Signature Type	32 bytes
Signature Size	4 bytes
Signature	256 bytes

Signature Type

The signature type is a string that represents the cryptographic algorithm being used and serves as a marker for the trailer. The bootloader supports the elliptic-curve digital signature algorithm (ECDSA). The default is sig-sha256-ecdsa.

Signature Size

The size of the cryptographic signature, in bytes.

Signature

The cryptographic signature of the application binary prepended with the image descriptor.

Bootloader configuration

The basic bootloader configuration options are provided in *freertos*`/vendors/microchip/boards/curiosity_pic32mzef/bootloader/config_files/aws_boot_config.h`. Some options are provided for debugging purposes only.

Enable Default Start

Enables the execution of the application from the default address and must be enabled for debugging only. The image is executed from the default address without any verification.

Enable Crypto Signature Verification

Enables cryptographic signature verification on boot. Failed images are erased from the device. This option is provided for debugging purposes only and must remain enabled in production.

Erase Invalid Image

Enables a full bank erase if image verification on that bank fails. The option is provided for debugging and must remain enabled in production.

Enable Hardware ID Verification

Enables verification of the hardware ID in the descriptor of the OTA image and the hardware ID programmed in the bootloader. This is optional and can be disabled if hardware ID verification is not required.

Enable Address Verification

Enables verification of the start, end, and execution addresses in the descriptor of OTA image. We recommend that you keep this option enabled.

Building the bootloader

The demo bootloader is included as a loadable project in the `aws_demos` project located in *`freertos`*`/vendors/microchip/boards/curiosity_pic32mzef/aws_demos/mplab/` in the FreeRTOS source code repository. When the `aws_demos` project is built, it builds the bootloader first, followed by the application. The final output is a unified hex image including the bootloader and the application. The `factory_image_generator.py` utility is provided to generate a unified hex image with cryptographic signature. The bootloader utility scripts are located in *`freertos`*`/demos/ota/bootloader/utility/`.

Bootloader pre-build step

This pre-build step executes a utility script called `codesigner_cert_utility.py` that extracts the public key from the code-signing certificate and generates a C header file that contains the public key in Abstract Syntax Notation One (ASN.1) encoded format. This header is compiled into the bootloader project. The generated header contains two constants: an array of the public key and the length of the key. The bootloader project can also be built without `aws_demos` and can be debugged as a normal application.

AWS IoT Device Defender demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

This demo shows you how to use the AWS IoT Device Defender library to connect to AWS IoT Device Defender. The demo uses the coreMQTT library to establish an MQTT connection over TLS (mutual authentication) to the AWS IoT MQTT Broker and the coreJSON library to validate and parse responses received from the AWS IoT Device Defender service. The demo shows you how to construct a JSON formatted report using metrics collected from the device, and how to submit the constructed report to the AWS IoT Device Defender service. The demo also shows how to register a callback function with the coreMQTT library to handle the responses from the AWS IoT Device Defender service to confirm whether a sent report was accepted or rejected.

Note

To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Functionality

This demo creates a single application task that demonstrates how to collect metrics, construct a device defender report in JSON format, and submit it to the AWS IoT Device Defender service through a secure MQTT connection to the AWS IoT MQTT Broker. The demo includes the standard networking metrics as well as custom metrics. For custom metrics, the demo includes:

- A metric named "`task_numbers`" which is a list of FreeRTOS task IDs. The type of this metric is "list of numbers".
- A metric named "`stack_high_water_mark`" which is the stack high watermark for the demo application task. The type of this metric is "number".

How we collect networking metrics depends on the TCP/IP stack in use. For FreeRTOS+TCP and supported lwIP configurations, we provide metrics collection implementations that collect real metrics from the device and submit them in the AWS IoT Device Defender report. You can find the implementations for FreeRTOS+TCP and lwIP on GitHub.

For boards using any other TCP/IP stack, we provide stub definitions of the metrics collection functions that return zeros for all networking metrics. Implement the functions in *`freertos`*`/demos/device_defender_for_aws/metrics_collector/stub/metrics_collector.c` for your network stack to send real metrics. The file is also available on the GitHub website.

For ESP32, the default lwIP configuration does not use core locking and therefore the demo will use stubbed metrics. If you want to use the reference lwIP metrics collection implementation, define the following macros in `lwiopts.h`:

```
#define LINK_SPEED_OF_YOUR_NETIF_IN_BPS 0
#define LWIP_TCPIP_CORE_LOCKING         1
#define LWIP_STATS                      1
#define MIB2_STATS                      1
```

The following is an example output when you run the demo.

```
24 1682 [iot_thread] [INFO] MQTT connection successfully established with broker.


25 1682 [iot_thread] [INFO] A clean MQTT connection is established. Cleaning up all the stored outgoing publishes.


26 1682 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/accepted.27 1682 [
iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.
28 1722 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
29 1722 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


30 1722 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/accepted with maximum QoS 1.
31 2322 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/rejected.32 2322 [
iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.
33 2382 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
34 2382 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


35 2382 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/rejected with maximum QoS 1.
36 2982 [iot_thread] [INFO] the published payload:{"hed": {"rid": 1109,"v": "1.0"},"met": {"tp": {"pts": [{"pt": 33251}],"t": 1},
"up": {"pts": [],"t": 0},"ns": {"bi": 0,"bo": 0,"pi": 0,"po": 0},"tc": {"ec": {"cs": [{"lp": 33251,"rad": "44.236.152.27:888337 2
982 [iot_thread] [INFO] PUBLISH sent for topic $aws/things/DemoThing/defender/metrics/json to broker with packet ID 3.


38 3102 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
39 3102 [iot_thread] [INFO] Ack packet deserialized with result: MQTTSuccess.
40 3102 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
41 3102 [iot_thread] [INFO] PUBACK received for packet id 3.


42 3102 [iot_thread] [INFO] Cleaned up outgoing publish packet with packet id 3.


43 3102 [iot_thread] [INFO] Packet received. ReceivedBytes=141.
44 3102 [iot_thread] [INFO] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
45 3102 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
46 3502 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.


47 3542 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
48 3542 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


49 4142 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.


50 4202 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
51 4202 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


52 4802 [iot_thread] [INFO] Disconnected from the broker.
53 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::freertos_heap::before::bytes::2088152

54 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::freertos_heap::after::bytes::1985556

55 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::demo_task_stack::before::bytes::1908

56 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::demo_task_stack::after::bytes::1908

57 5802 [iot_thread] [INFO ][DEMO][5802] Demo completed successfully.

58 5804 [iot_thread] [INFO ][INIT][5804] SDK cleanup done.

59 5804 [iot_thread] [INFO ][DEMO][5804] -------DEMO FINISHED-------
```

If your board isn't using FreeRTOS+TCP or a supported lwIP configuration, the output will look like the following.

```
24 1716 [iot_thread] [INFO] MQTT connection successfully established with broker.


25 1716 [iot_thread] [INFO] A clean MQTT connection is established. Cleaning up all the stored outgoing publishes.


26 1716 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/accepted.27 1716
 [iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.
28 1756 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
29 1756 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


30 1756 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/accepted with maximum QoS 1.
31 2356 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/rejected.32 2356
 [iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.
33 2436 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
34 2436 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


35 2436 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/rejected with maximum QoS 1.
36 3036 [iot_thread] [ERROR] Using stub definition of GetNetworkStats! Please implement for your network stack to get correct m
etrics.
37 3036 [iot_thread] [ERROR] Using stub definition of GetOpenTcpPorts! Please implement for your network stack to get correct m
etrics.
38 3036 [iot_thread] [ERROR] Using stub definition of GetOpenUdpPorts! Please implement for your network stack to get correct m
etrics.
39 3036 [iot_thread] [ERROR] Using stub definition of GetEstablishedConnections! Please implement for your network stack to get
 correct metrics.
40 3036 [iot_thread] [INFO] the published payload:{"hed": {"rid": 1079,"v": "1.0"},"met": {"tp": {"pts": [],"t": 0},"up": {"pts
": [],"t": 0},"ns": {"bi": 0,"bo": 0,"pi": 0,"po": 0},"tc": {"ec": {"cs": [],"t": 0}}},"cmet": {"stack_high_water_mark": [{"num
41 3036 [iot_thread] [INFO] PUBLISH sent for topic $aws/things/DemoThing/defender/metrics/json to broker with packet ID 3.


42 3196 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
43 3196 [iot_thread] [INFO] Ack packet deserialized with result: MQTTSuccess.
44 3196 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
45 3196 [iot_thread] [INFO] PUBACK received for packet id 3.


46 3196 [iot_thread] [INFO] Cleaned up outgoing publish packet with packet id 3.


47 3196 [iot_thread] [INFO] Packet received. ReceivedBytes=141.
48 3196 [iot_thread] [INFO] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
49 3196 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
50 3596 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.


51 3656 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
52 3656 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


53 4256 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.


54 4336 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
55 4336 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


56 4936 [iot_thread] [INFO] Disconnected from the broker.
57 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::freertos_heap::before::bytes::2088152

58 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::freertos_heap::after::bytes::1985556

59 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::demo_task_stack::before::bytes::1908

60 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::demo_task_stack::after::bytes::1908

61 5936 [iot_thread] [INFO ][DEMO][5936] Demo completed successfully.

62 5938 [iot_thread] [INFO ][INIT][5938] SDK cleanup done.

63 5938 [iot_thread] [INFO ][DEMO][5938] -------DEMO FINISHED-------
```

The source code of the demo is in your download in *freertos*/demos/device_defender_for_aws/ directory or on the GitHub website.

Subscribing to AWS IoT Device Defender topics

The subscribeToDefenderTopics function subscribes to the MQTT topics on which responses to published Device Defender reports will be received. It uses the macro `DEFENDER_API_JSON_ACCEPTED` to construct the topic string on which responses for accepted device defender reports are received. It uses the macro `DEFENDER_API_JSON_REJECTED` to construct the topic string on which responses for rejected device defender reports will be received.

Collecting device metrics

The collectDeviceMetrics function gathers networking metrics using the functions defined in `metrics_collector.h`. The metrics collected are the number of bytes and packets sent and received, the open TCP ports, the open UDP ports, and the established TCP connections.

Generating the AWS IoT Device Defender report

The generateDeviceMetricsReportfunction generates a device defender report using the function defined in `report_builder.h`. That function takes the networking metrics and a buffer, creates a JSON document in the format as expected by AWS IoT Device Defender and writes it to the provided buffer. The format of the JSON document expected by AWS IoT Device Defender is specified in Device-side metrics in the *AWS IoT Developer Guide*.

Publishing the AWS IoT Device Defender report

The AWS IoT Device Defender report is published on the MQTT topic for publishing JSON AWS IoT Device Defender reports. The report is constructed using the macro `DEFENDER_API_JSON_PUBLISH`, as shown in this code snippet on the GitHub website.

Callback for handling responses

The publishCallback function handles incoming MQTT publish messages. It uses the `Defender_MatchTopic` API from the AWS IoT Device Defender library to check if the incoming MQTT message is from the AWS IoT Device Defender service. If the message is from the AWS IoT Device Defender service, it parses the received JSON response and extracts the report ID in the response. The report ID is then verified to be the same as the one sent in the report.

AWS IoT Greengrass V1 discovery demo application

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Before you run the AWS IoT Greengrass Discovery demo for FreeRTOS, you need to set up AWS, AWS IoT Greengrass, and AWS IoT. To set up AWS, follow the instructions at Setting up your AWS account and permissions. To set up AWS IoT Greengrass, you need to create a Greengrass group and then add a Greengrass core. For more information about setting up AWS IoT Greengrass, see Getting Started with AWS IoT Greengrass.

After you set up AWS and AWS IoT Greengrass, you need to configure some additional permissions for AWS IoT Greengrass.

To set up AWS IoT Greengrass permissions

1. Browse to the IAM console.
2. From the navigation pane, choose **Roles**, and then find and choose **Greengrass_ServiceRole**.
3. Choose **Attach policies**, select **AmazonS3FullAccess** and **AWSIoTFullAccess**, and then choose **Attach policy**.
4. Browse to the AWS IoT console.
5. In the navigation pane, choose **Greengrass**, choose **Groups**, and then choose the Greengrass group that you previously created.
6. Choose **Settings**, and then choose **Add role**.
7. Choose **Greengrass_ServiceRole**, and then choose **Save**.

Connect your board to AWS IoT and configure your FreeRTOS demo.

1. Registering your MCU board with AWS IoT

   After you register your board, you need to create and attach a new Greengrass policy to the device's certificate.

To create a new AWS IoT Greengrass 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:

   ```
   {
         "Effect": "Allow",
         "Action": [
           "greengrass:*"
         ],
         "Resource": "*"
   }
   ```

   This policy grants AWS IoT Greengrass permissions to all resources.
5. Choose **Create**.

To attach the AWS IoT Greengrass policy to your device's certificate

1. Browse to the AWS IoT console.
2. In the navigation pane, choose **Manage**, choose **Things**, and then choose the thing that you previously created.
3. Choose **Security**, and then choose the certificate attached to your device.
4. Choose **Policies**, choose **Actions**, and then choose **Attach Policy**.
5. Find and choose the Greengrass policy that you created earlier, and then choose **Attach**.

2. Downloading FreeRTOS

> **Note**
>
> If you are downloading FreeRTOS from the FreeRTOS console, choose **Connect to AWS IoT Greengrass-** ***Platform*** instead of **Connect to AWS IoT-** ***Platform***.

3. Configuring the FreeRTOS demos.

Open *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h, comment out #define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED, and define CONFIG_GREENGRASS_DISCOVERY_DEMO_ENABLED.

After you set up AWS IoT and AWS IoT Greengrass, and after you download and configure FreeRTOS, you can build, flash, and run the Greengrass demo on your device. To set up your board's hardware and software development environment, follow the instructions in the Board-specific getting started guides.

The Greengrass demo publishes a series of messages to the Greengrass core, and to the AWS IoT MQTT client. To view the messages in the AWS IoT MQTT client, open the AWS IoT console, choose **Test**, choose **MQTT test client** and then add a subscription to freertos/demos/ggd.

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

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

Using an Amazon EC2 instance

If you are working with an Amazon EC2 instance

1. Find the Public DNS (IPv4) associated with your Amazon EC2 instance— go to the Amazon EC2 console, and in the left navigation panel, choose **Instances**. Choose your Amazon EC2 instance, and then choose the **Description** panel. Look for the entry for the **Public DNS (IPv4)** and make a note of it.
2. Find the entry for **Security groups** and choose the security group attached to your Amazon EC2 instance.
3. Choose the **Inbound rules** tab then choose **Edit inbound rules** and add the following rules.

Inbound rules

Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	0.0.0.0/0	-
HTTP	TCP	80	::/0	-
SSH	TCP	22	0.0.0.0/0	-
Custom TCP	TCP	8883	0.0.0.0/0	MQTT communications
Custom TCP	TCP	8883	::/0	MQTT communications
HTTPS	TCP	443	0.0.0.0/0	-
HTTPS	TCP	443	::0/0	-
All ICMP - IPv4	ICMP	All	0.0.0.0/0	-
All ICMP - IPv4	ICMP	All	::0/0	-

4. In the AWS IoT console choose **Greengrass**, then **Groups**, and choose the Greengrass group that you previously created. Choose **Settings**. Change the **Local connection detection** to **Manually manage connection information**.
5. In the navigation pane, choose **Cores** then select your group core.
6. Choose **Connectivity** and make sure you have only one core endpoint (delete all of the rest) and that it is not an IP address (because it is subject to change). The best option is to use the Public DNS (IPv4) that you noted in the first step.
7. Add the FreeRTOS IoT thing you created to the GG group.

    a. Choose the back arrow to return to the AWS IoT Greengrass group page. In the navigation pane, choose **Devices** then choose **Add Device**.

    b. Choose **Select an IoT Thing**. Choose your device then choose **Finish**.

8. Add the necessary subscriptions— in the **Greengrass Group** page, choose **Subscriptions** then choose **Add Subscription** and enter information as shown here.

   Subscriptions

Source	Target	Topic
TIGG1	IoT Cloud	freertos/demos/ggd

   Where "Source" is the name given to the AWS IoT thing created in the AWS IoT console when you registered your board - "TIGG1" in the example given here.

9. Start a deployment of your AWS IoT Greengrass group and make sure that the deployment is successful. You should now be able to successfully run the AWS IoT Greengrass discovery demo.

AWS IoT Greengrass V2

Compatibility with AWS IoT Greengrass V2 devices

AWS IoT Greengrass V2 support for client devices is backwards-compatible with AWS IoT Greengrass V1. You can connect FreeRTOS client devices to V2 core devices without changing the application code. To enable client devices to connect to a V2 core device, do the following.

- Deploy Greengrass software to the Greengrass core device. See Connect client devices to core devices to connect a device to AWS IoT Greengrass V2.
- To relay messages (including Lambda functions) between client devices, AWS IoT Core cloud service, and Greengrass components, deploy and configure the MQTT bridge component.
- Deploy the IP detector component to automatically detect connectivity information, or manually manage the endpoints.
- See Interact with local AWS IoT devices for more information.

For more details, see AWS documentation about running AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2.

coreHTTP demos

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

These demos can help you learn how to use the coreHTTP library.

Topics

- coreHTTP mutual authentication demo
- coreHTTP basic Amazon S3 upload demo
- coreHTTP basic S3 download demo
- coreHTTP basic multithreaded demo

coreHTTP mutual authentication demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

The coreHTTP (Mutual Authentication) demo project shows you how to establish a connection to an HTTP server using TLS with mutual authentication between the client and the server. This demo uses an mbedTLS-based transport interface implementation to establish a server- and client-authenticated TLS connection, and demonstrates a request response workflow in HTTP.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Functionality

This demo creates a single application task with examples that show how to complete the following:

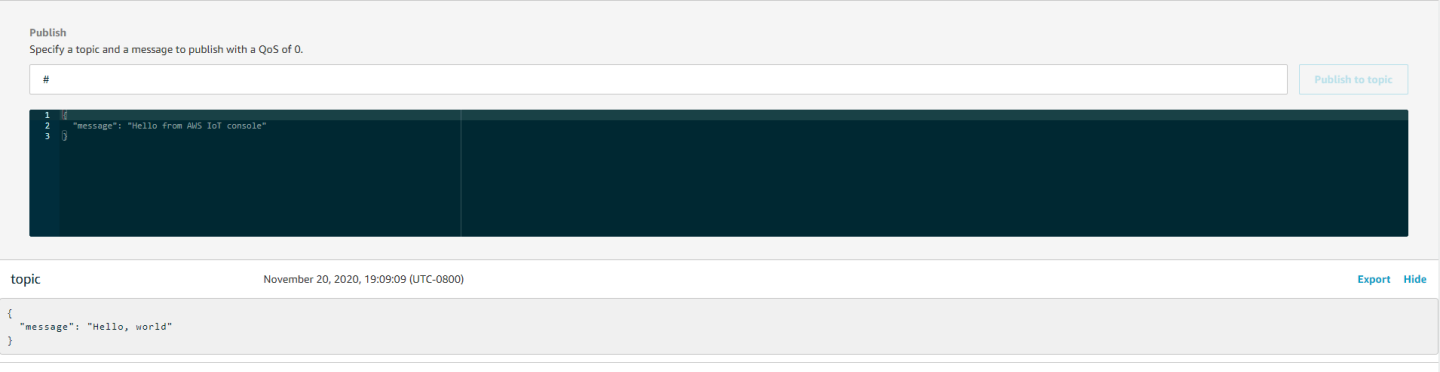
- Connect to the HTTP server on the AWS IoT endpoint.
- Send a POST request.
- Receive the response.
- Disconnect from the server.

After you complete these steps, the demo generates output similar to the following screenshot.

```
9 1565 [iot_thread] [INFO ][DEMO][1565] ---------STARTING DEMO---------

10 1566 [iot_thread] [INFO ][INIT][1566] SDK successfully initialized.
11 1566 [iot_thread] [INFO ][DEMO][1566] Successfully initialized the demo. Network type for the demo: 4
12 1566 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:319] 13 1566 [iot_thread] Establishing a TLS session to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com:8443.14 1566 [iot_thread]
15 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (35.166.102.88) will be stored
16 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (35.166.2.12) will be stored
17 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (44.239.154.164) will be stored
18 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (44.239.97.33) will be stored
19 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (44.238.43.70) will be stored
20 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (52.32.144.134) will be stored
21 2042 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:393] 22 2042 [iot_thread] Sending HTTP POST request to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com/topics/topic?qos=1...23 2042 [iot_thread]
24 2082 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:418] 25 2082 [iot_thread] Received HTTP response from a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com/topics/topic?qos=1...
26 2082 [iot_thread]
27 2082 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:283] 28 2082 [iot_thread] Demo completed successfully.29 2082 [iot_thread]
30 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::freertos_heap::before::bytes::2088152
31 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::freertos_heap::after::bytes::1990104
32 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::demo_task_stack::before::bytes::1908
33 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::demo_task_stack::after::bytes::1908
34 3082 [iot_thread] [INFO ][DEMO][3082] Demo completed successfully.
35 3084 [iot_thread] [INFO ][INIT][3084] SDK cleanup done.
36 3084 [iot_thread] [INFO ][DEMO][3084] -------DEMO FINISHED-------
```

The AWS IoT console generates output similar to the following screenshot.



Source code organization

The demo source file is named `http_demo_mutual_auth.c` and can be found in the *`freertos`*`/demos/coreHTTP/` directory and on the GitHub website.

Connecting to the AWS IoT HTTP server

The connectToServerWithBackoffRetries function attempts to make a mutually authenticated TLS connection to the AWS IoT HTTP server. If the connection fails, it retries after a timeout. The timeout value exponentially increases until the maximum number of attempts is reached or the maximum timeout value is reached. The `RetryUtils_BackoffAndSleep` function provides exponentially increasing timeout values and returns `RetryUtilsRetriesExhausted` when the maximum number of attempts have been reached. The `connectToServerWithBackoffRetries` function returns a failure status if the TLS connection to the broker can't be established after the configured number of attempts.

Sending an HTTP request and receiving the response

The prvSendHttpRequest function demonstrates how to send a POST request to the AWS IoT HTTP server. For more information on making a request to the REST API in AWS IoT, see Device communication protocols - HTTPS. The response is received with the same coreHTTP API call, `HTTPClient_Send`.

coreHTTP basic Amazon S3 upload demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

This example demonstrates how to send a PUT request to the Amazon Simple Storage Service (Amazon S3) HTTP server and upload a small file. It also performs a GET request to verify the size of the file after the upload. This example uses a network transport interface that uses mbedTLS to establish a mutually authenticated connection between an IoT device client running coreHTTP and the Amazon S3 HTTP server.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Single threaded versus multi threaded

There are two coreHTTP usage models, *single threaded* and *multithreaded* (multitasking). Although the demo in this section runs the HTTP library in a thread, it actually demonstrates how to use coreHTTP in a single threaded environment. Only one task in this demo uses the HTTP API. Although single threaded applications must repeatedly call the HTTP library, multithreaded applications can instead send HTTP requests in the background within an agent (or daemon) task.

Source code organization

The demo source file is named `http_demo_s3_upload.c` and can be found in the *`freertos`*`/demos/coreHTTP/` directory and on the GitHub website.

Configuring the Amazon S3 HTTP server connection

This demo uses a pre-signed URL to connect to the Amazon S3 HTTP server and authorize access to the object to download. The Amazon S3 HTTP server's TLS connection uses server authentication only. At the application level, access to the object is authenticated with parameters in the pre-signed URL query. Follow the steps below to configure your connection to AWS.

1. Set up an AWS account:
   a. If you haven't already, create an AWS account.
   b. Accounts and permissions are set using AWS Identity and Access Management (IAM). You use IAM to manage permissions for each user in your account. By default, a user doesn't have permissions until granted by the root owner.
      i. To add a user to your AWS account, see the IAM User Guide.
      ii. Grant permission to your AWS account to access FreeRTOS and AWS IoT by adding this policy:
         - AmazonS3FullAccess
2. Create a bucket in Amazon S3 by following the steps in How do I create an S3 bucket? in the *Amazon Simple Storage Service User Guide*.

3. Upload a file to Amazon S3 by following the steps in How do I upload files and folders to an S3 bucket?.
4. Generate a pre-signed URL using the script located at the `FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/presigned_urls_gen.py` file.

   For usage instructions, see the `FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/README.md` file.

Functionality

The demo first connects to the Amazon S3 HTTP server with TLS server authentication. Then, it creates an HTTP request to upload the data specified in `democonfigDEMO_HTTP_UPLOAD_DATA`. After uploading the file, it checks that file was successfully uploaded by requesting for the size of the file. The source code for the demo can be found on the GitHub website.

Connecting to the Amazon S3 HTTP server

The connectToServerWithBackoffRetries function attempts to make a TCP connection to the HTTP server. If the connection fails, it retries after a timeout. The timeout value will exponentially increase until the maximum number of attempts are reached or the maximum timeout value is reached. The `connectToServerWithBackoffRetries` function returns a failure status if the TCP connection to the server can't be established after the configured number of attempts.

The `prvConnectToServer` function demonstrates how to establish a connection to the Amazon S3 HTTP server by using server authentication only. It uses the mbedTLS-based transport interface that is implemented in the `FreeRTOS-Plus/Source/Application-Protocols/network_transport/freertos_plus_tcp/using_mbedtls/using_mbedtls.c` file. The definition of `prvConnectToServer` can be found on the GitHub website.

Upload data

The `prvUploadS3ObjectFile` function demonstrates how to create a PUT request and specify the file to upload. The Amazon S3 bucket where the file is uploaded and the name of file to upload are specified in the pre-signed URL. To save memory, the same buffer is used for both the request headers and to receive the response. The response is received synchronously using the `HTTPClient_Send` API function. A `200 OK` response status code is expected from the Amazon S3 HTTP server. Any other status code is an error.

The source code for `prvUploadS3ObjectFile()` can be found on the GitHub website.

Verifying the upload

The `prvVerifyS3ObjectFileSize` function calls `prvGetS3ObjectFileSize` to retrieve the size of the object in the S3 bucket. The Amazon S3 HTTP server doesn't currently support HEAD requests using a pre-signed URL, so the 0th byte is requested. The size of the file is contained in the response's `Content-Range` header field. A `206 Partial Content` response is expected from the server. Any other response status code is an error.

The source code for `prvGetS3ObjectFileSize()` can be found on the GitHub website.

coreHTTP basic S3 download demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

This demo shows how to use range requests to download files from the Amazon S3 HTTP server. Range requests are natively supported in the coreHTTP API when you use `HTTPClient_AddRangeHeader` to create the HTTP request. For a microcontroller environment, range requests are highly encouraged. By downloading a large file in separate ranges, instead of in a single request, each section of the file can be processed without blocking the network socket. Range requests lower the risk of having dropped packets, which require retransmissions on the TCP connection, and so they improve the power consumption of the device.

This example uses a network transport interface that uses mbedTLS to establish a mutually authenticated connection between an IoT device client running coreHTTP and the Amazon S3 HTTP server.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Single threaded versus multi threaded

There are two coreHTTP usage models, *single threaded* and *multithreaded* (multitasking). Although the demo in this section runs the HTTP library in a thread, it actually demonstrates how to use coreHTTP in a single threaded environment (only one task uses the HTTP API in the demo). Although single threaded applications must repeatedly call the HTTP library, multithreaded applications can instead send HTTP requests in the background within an agent (or daemon) task.

Source code organization

The demo project is named `http_demo_s3_download.c` and can be found in the *`freertos`*`/demos/coreHTTP/` directory and on the GitHub website.

Configuring the Amazon S3 HTTP server connection

This demo uses a pre-signed URL to connect to the Amazon S3 HTTP server and authorize access to the object to download. The Amazon S3 HTTP server's TLS connection uses server authentication only. At the application level, access to the object is authenticated with parameters in the pre-signed URL query. Follow the steps below to configure your connection to AWS.

1. Set up an AWS account:
    a. If you haven't already, create and activate an AWS account.
    b. Accounts and permissions are set using AWS Identity and Access Management (IAM). IAM allows you to manage permissions for each user in your account. By default, a user doesn't have permissions until granted by the root owner.
        i. To add a user to your AWS account, see the IAM User Guide.
        ii. Grant permission to your AWS account to access FreeRTOS and AWS IoT by adding these policies:
            - AmazonS3FullAccess
2. Create a bucket in S3 by following the steps in How do I create an S3 Bucket? in the *Amazon Simple Storage Service Console User Guide*.
3. Upload a file to S3 by following the steps in How do I upload files and folders to an S3 bucket?.
4. Generate a pre-signed URL using the script located at `FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/`

`presigned_urls_gen.py`. For usage instructions, see `FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/README.md`.

Functionality

The demo retrieves the size of the file first. Then it requests each byte range sequentially, in a loop, with range sizes of `democonfigRANGE_REQUEST_LENGTH`.

Source code for the demo can be found on the GitHub website.

Connecting to the Amazon S3 HTTP server

The function `connectToServerWithBackoffRetries()` attempts to make a TCP connection to the HTTP server. If the connection fails, it retries after a timeout. The timeout value will exponentially increase until the maximum number of attempts are reached or the maximum timeout value is reached. `connectToServerWithBackoffRetries()` returns a failure status if the TCP connection to the server cannot be established after the configured number of attempts.

The function `prvConnectToServer()` demonstrates how to establish a connection to the Amazon S3 HTTP server using server authentication only. It uses the mbedTLS-based transport interface that is implemented in the file `FreeRTOS-Plus/Source/Application-Protocols/network_transport/freertos_plus_tcp/using_mbedtls/using_mbedtls.c`.

The source code for `prvConnectToServer()` can be found on GitHub.

Creating a range request

The API function `HTTPClient_AddRangeHeader()` supports serializing a byte range into the HTTP request headers to form a range request. Range requests are used in this demo to retrieve the file size and to request each section of the file.

The function `prvGetS3ObjectFileSize()` retrieves the size of the file in the S3 bucket. The `Connection: keep-alive` header is added in this first request to Amazon S3 to keep the connection open after the response is sent. The S3 HTTP server does not currently support HEAD requests using a pre-signed URL, so the 0th byte is requested. The size of the file is contained in the response's `Content-Range` header field. A `206 Partial Content` response is expected from the server; any other response status-code received is an error.

The source code for `prvGetS3ObjectFileSize()` can be found on GitHub.

After it retrieves the file size, this demo creates a new range request for each byte range of the file to download. It uses `HTTPClient_AddRangeHeader()` for each section of the file.

Sending range requests and receiving responses

The function `prvDownloadS3ObjectFile()` sends the range requests in a loop until the entire file is downloaded. The API function `HTTPClient_Send()` sends a request and receives the response synchronously. When the function returns, the response is received in an `xResponse`. The status-code is then verified to be `206 Partial Content` and the number of bytes downloaded so far is incremented by the `Content-Length` header value.

The source code for `prvDownloadS3ObjectFile()` can be found on GitHub.

coreHTTP basic multithreaded demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

This demo uses FreeRTOS's thread-safe queues to hold requests and responses waiting to be processed. In this demo, there are three tasks to take note of.

- The main task waits for requests to appear in the request queue. It will send those requests over the network, then place the response into the response queue.
- A request task creates HTTP library request objects to send to the server and places them into the request queue. Each request object specifies a byte range of the S3 file that the application has configured for download.
- A response task waits for responses to appear in the response queue. It logs every response it receives.

This basic multithreaded demo is configured to use a TLS connection with server authentication only, this is required by the Amazon S3 HTTP server. Application layer authentication is done using the Signature Version 4 parameters in the presigned URL query.

Source code organization

The demo project is named `http_demo_s3_download_multithreaded.c` and can be found in the *`freertos`*`/demos/coreHTTP/` directory and the GitHub website.

Building the demo project

The demo project uses the free community edition of Visual Studio. To build the demo:

1. Open the `mqtt_multitask_demo.sln` Visual Studio solution file from within the Visual Studio IDE.
2. Select **Build Solution** from the IDE's **Build** menu.

> **Note**
>
> If you are using Microsoft Visual Studio 2017 or earlier, then you must select a **Platform Toolset** compatible with your version: **Project -> RTOSDemos Properties -> Platform Toolset**.

Configuring the demo project

The demo uses the FreeRTOS+TCP TCP/IP stack, so follow the instructions provided for the TCP/IP starter project to:

1. Install the pre-requisite components (such as WinPCap).
2. Optionally set a static or dynamic IP address, gateway address and netmask.
3. Optionally set a MAC address.
4. Select an Ethernet network interface on your host machine.
5. **Importantly** test your network connection before attempting to run the HTTP demo.

Configuring the Amazon S3 HTTP server connection

Follow the instructions for Configuring the Amazon S3 HTTP server connection in the *coreHTTP basic download demo*.

Functionality

The demo creates three tasks in total:

- One that sends requests and receives responses over the network.
- One that creates requests to send.
- One that processes the received responses.

In this demo, the primary task:

1. Creates the request and response queues.
2. Creates the connection to the server.
3. Creates the request and response tasks.
4. Waits for the request queue to send requests over the network.
5. Places responses received over the network into the response queue.

The request task:

1. Creates each of the range requests.

The response task:

1. Processes each of the responses received.

Typedefs

The demo defines the following structures to support multithreading.

Request items

The following structures define a request item to place into the request queue. The request item is copied into the queue after the request task creates an HTTP request.

```
/**
 * @brief Data type for the request queue.
 *
 * Contains the request header struct and its corresponding buffer, to be
 * populated and enqueued by the request task, and read by the main task. The
 * buffer is included to avoid pointer inaccuracy during queue copy operations.
 */
```

```
typedef struct RequestItem
{
    HTTPRequestHeaders_t xRequestHeaders;
    uint8_t ucHeaderBuffer[ democonfigUSER_BUFFER_LENGTH ];
} RequestItem_t;
```

Response Item

The following structures define a response item to place into the response queue. The response item is copied into the queue after the main HTTP task receives a response over the network.

```

/**
 * @brief Data type for the response queue.
 *
 * Contains the response data type and its corresponding buffer, to be enqueued
 * by the main task, and interpreted by the response task. The buffer is
 * included to avoid pointer inaccuracy during queue copy operations.
 */
typedef struct ResponseItem
{
    HTTPResponse_t xResponse;
    uint8_t ucResponseBuffer[ democonfigUSER_BUFFER_LENGTH ];
} ResponseItem_t;
```

Main HTTP send task

The main application task:

1. Parses the presigned URL for the host address to establish a connection with the Amazon S3 HTTP server.
2. Parses the presigned URL for the path to the objects in the S3 bucket.
3. Connects to the Amazon S3 HTTP server using TLS with server authentication.
4. Creates the request and response queues.
5. Creates the request and response tasks.

The function `prvHTTPDemoTask()` does this set up, and gives the demo status. The source code for this function can be found on Github.

In the function `prvDownloadLoop()`, the main task blocks and waits on requests from the request queue. When it receives a request it sends it using API function `HTTPClient_Send()`. If the API function was successful, then it places the response into the response queue.

The source code for `prvDownloadLoop()` can be found on Github.

HTTP request task

The request task is specified in the function `prvRequestTask`. The source code for this function can be found on Github.

The request task retrieves the size of the file in the Amazon S3 bucket. This is done in the function `prvGetS3ObjectFileSize`. The "Connection: keep-alive" header is added to this request to Amazon S3 to keep the connection open after the response is sent. The Amazon S3 HTTP server does not currently support HEAD requests using a presigned URL, so the 0th byte is requested. The size of the file is contained in the response's `Content-Range` header field. A `206 Partial Content` response is expected from the server; any other response status-code received is an error.

The source code for `prvGetS3ObjectFileSize` can be found on Github.

After retrieving the file size, the request task continues to request each range of the file. Each range request is placed into the request queue for the main task to send. The file ranges are configured by the demo user in the macro `democonfigRANGE_REQUEST_LENGTH`. Range requests are natively supported in the HTTP client library API using the function `HTTPClient_AddRangeHeader`. The function `prvRequestS3ObjectRange` demonstrates how to use `HTTPClient_AddRangeHeader()`.

The source code for the function `prvRequestS3ObjectRange` can be found on Github.

HTTP response task

The response tasks waits on the response queue for responses received over the network. The main task populates the response queue when it successfully receives an HTTP response. This task processes the responses by logging the status-code, headers, and body. A real-world application may process the response by writing the response body to flash memory, for example. If the response status-code is not `206 partial content`, then the task notifies the main task that the demo should fail. The response task is specified in function `prvResponseTask`. The source code for this function can be found on Github.

AWS IoT Jobs library demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

The AWS IoT Jobs library demo shows you how to connect to the AWS IoT Jobs service through an MQTT connection, retrieve a job from AWS IoT, and process it on a device. The AWS IoT Jobs demo project uses the FreeRTOS Windows port, so it can be built and evaluated with the Visual Studio Community version on Windows. No microcontroller hardware is needed. The demo establishes a secure connection to the AWS IoT MQTT broker using TLS in the same manner as the coreMQTT mutual authentication demo.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Source code organization

The demo code is in the `jobs_demo.c` file and can be found on the GitHub website or in the *`freertos`*`/demos/jobs_for_aws/` directory.

Configure the AWS IoT MQTT broker connection

In this demo, you use an MQTT connection to the AWS IoT MQTT broker. This connection is configured in the same way as the coreMQTT mutual authentication demo.

Functionality

The demo shows the workflow used to receive jobs from AWS IoT and process them on a device. The demo is interactive and requires you to create jobs by using either the AWS IoT console or the AWS Command Line Interface (AWS CLI). For more information about creating a job, see create-job

in the *AWS CLI Command Reference*. The demo requires the job document to have an `action` key set to `print` to print a message to the console.

See the following format for this job document.

```
{
    "action": "print",
    "message": "ADD_MESSAGE_HERE"
}
```

You can use the AWS CLI to create a job as in the following example command.

```
aws iot create-job \
    --job-id t12 \
    --targets arn:aws:iot:region:123456789012:thing/device1 \
    --document '{"action":"print","message":"hello world!"}'
```

The demo also uses a job document that has the `action` key set to `publish` to republish the message to a topic. See the following format for this job document.

```
{
    "action": "publish",
    "message": "ADD_MESSAGE_HERE",
    "topic": "topic/name/here"
}
```

The demo loops until it receives a job document with the `action` key set to `exit` to exit the demo. The format for the job document is as follows.

```
{
    "action: "exit"
}
```

Entry point of the Jobs demo

The source code for the Jobs demo entry point function can be found on GitHub. This function performs the following operations:

1. Establish an MQTT connection using the helper functions in `mqtt_demo_helpers.c`.

2. Subscribe to the MQTT topic for the `NextJobExecutionChanged` API, using helper functions in `mqtt_demo_helpers.c`. The topic string is assembled earlier, using macros defined by the AWS IoT Jobs library.
3. Publish to the MQTT topic for the `StartNextPendingJobExecution` API, using helper functions in `mqtt_demo_helpers.c`. The topic string is assembled earlier, using macros defined by the AWS IoT Jobs library.
4. Repeatedly call `MQTT_ProcessLoop` to receive incoming messages which are handed to `prvEventCallback` for processing.
5. After the demo receives the exit action, unsubscribe from the MQTT topic and disconnect, using the helper functions in the `mqtt_demo_helpers.c` file.

Callback for received MQTT messages

The prvEventCallback function calls `Jobs_MatchTopic` from the AWS IoT Jobs library to classify the incoming MQTT message. If the message type corresponds to a new job, `prvNextJobHandler()` is called.

The prvNextJobHandler function, and the functions it calls, parse the job document from the JSON-formatted message, and run the action specified by the job. Of particular interest is the `prvSendUpdateForJob` function.

Send an update for a running job

The function prvSendUpdateForJob() calls `Jobs_Update()` from the Jobs library to populate the topic string used in the MQTT publish operation that immediately follows.

coreMQTT demos

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

These demos can help you learn how to use the coreMQTT library.

Topics

- coreMQTT mutual authentication demo
- coreMQTT Agent connection sharing demo

coreMQTT mutual authentication demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

The coreMQTT mutual authentication demo project shows you how to establish a connection to an MQTT broker using TLS with mutual authentication between the client and the server. This demo uses an mbedTLS-based transport interface implementation to establish a server and client-authenticated TLS connection, and demonstrates the subscribe-publish workflow of MQTT at QoS 1 level. It subscribes to a topic filter, then publishes to topics that match the filter and waits for receipt of those messages back from the server at QoS 1 level. This cycle of publishing to the broker and receiving the same message back from the broker is repeated indefinitely. Messages in this demo are sent at QoS 1, which guarantees at least one delivery according to the MQTT spec.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

Source code

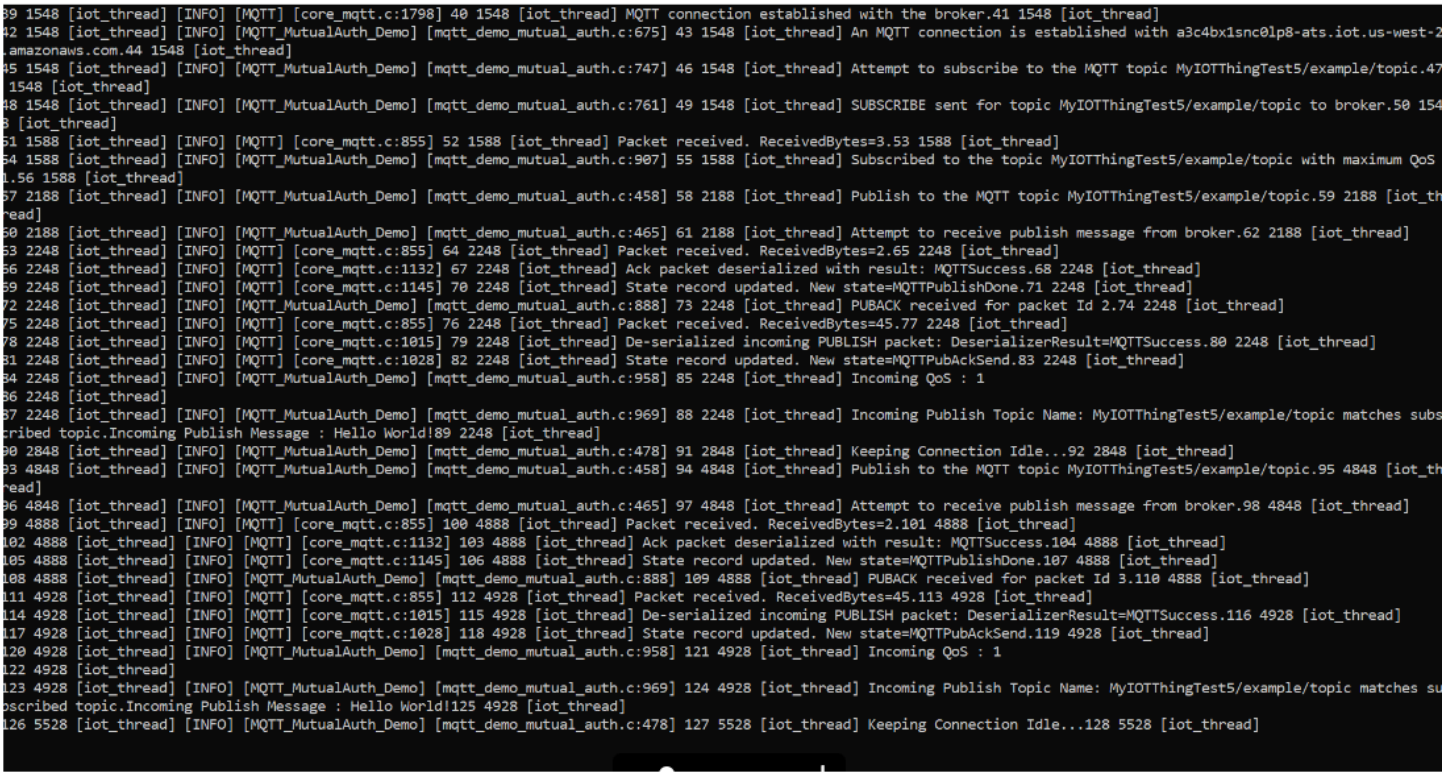
The demo source file is named `mqtt_demo_mutual_auth.c` and can be found in the *`freertos`*`/demos/coreMQTT/` directory and the GitHub website.

Functionality

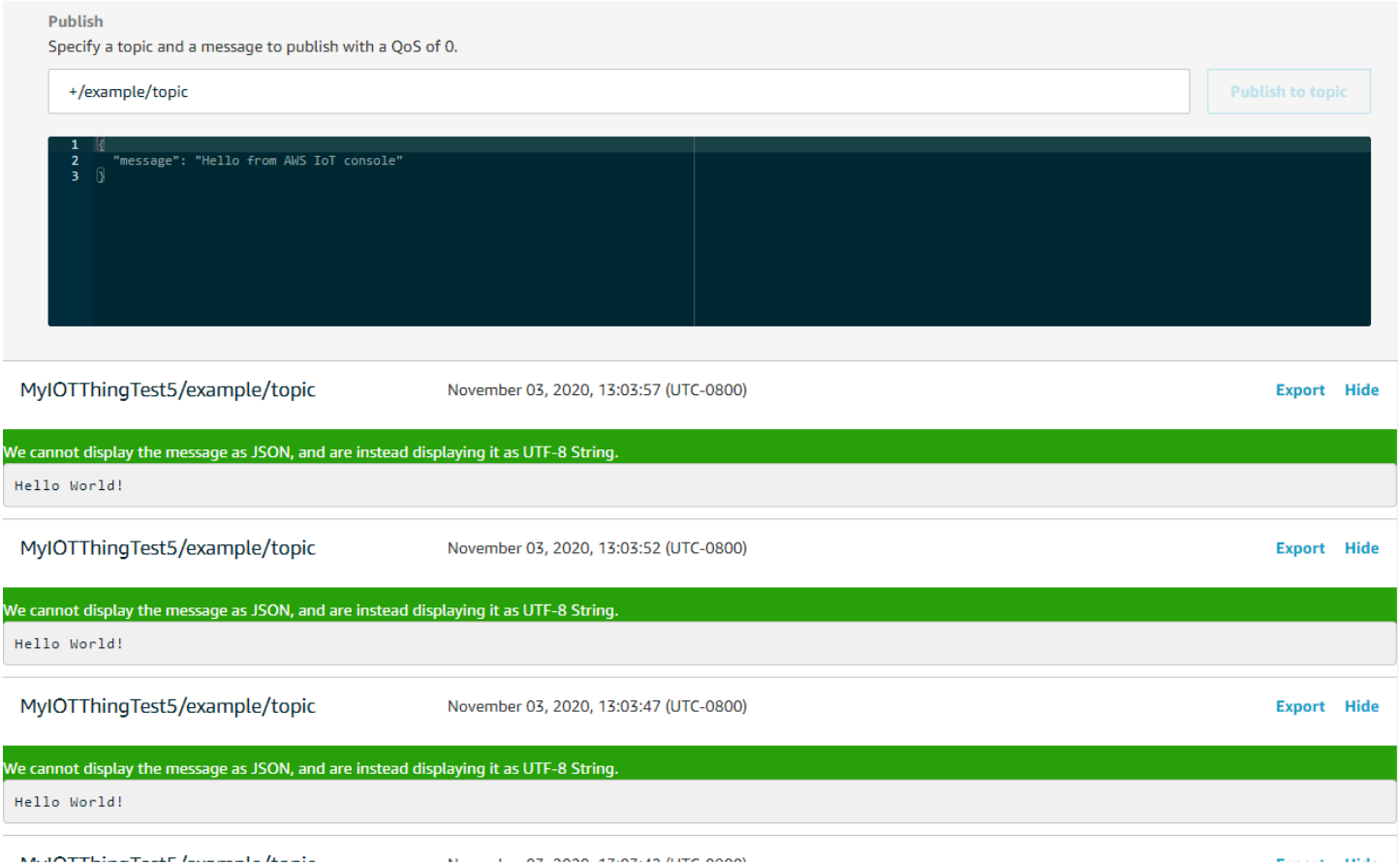
The demo creates a single application task that loops through a set of examples that demonstrate how to connect to the broker, subscribe to a topic on the broker, publish to a topic on the broker,

then finally, disconnect from the broker. The demo application both subscribes to and publishes to the same topic. Each time the demo publishes a message to the MQTT broker, the broker sends the same message back to the demo application.

A successful completion of the demo will generate an output similar to the following image.



The AWS IoT console will generate an output similar to the following image.



Retry logic with exponential backoff and jitter

The prvBackoffForRetry function shows how failed network operations with the server, for example, TLS connections or MQTT subscribe requests, can be retried with exponential backoff and jitter. The function calculates the backoff period for the next retry attempt, and performs the backoff delay if the retry attempts haven't been exhausted. Because the calculation of the backoff period requires the generation of a random number, the function uses the PKCS11 module to generate the random number. Use of the PKCS11 module allows access to a True Random Number Generator (TRNG) if the vendor platform supports it. We recommended that you seed the random number generator with a device-specific entropy source so that the probability of collisions from devices during connection retries is mitigated.

Connecting to the MQTT broker

The prvConnectToServerWithBackoffRetries function attempts to make a mutually authenticated TLS connection to the MQTT broker. If the connection fails, it retries after a backoff period. The backoff period will exponentially increase until the maximum number of attempts is reached or the maximum backoff period is reached. The

`BackoffAlgorithm_GetNextBackoff` function provides an exponentially increasing backoff value and returns `RetryUtilsRetriesExhausted` when the maximum number of attempts has been reached. The `prvConnectToServerWithBackoffRetries` function returns a failure status if the TLS connection to the broker can't be established after the configured number of attempts.

The `prvCreateMQTTConnectionWithBroker` function demonstrates how to establish an MQTT connection to an MQTT broker with a clean session. It uses the TLS transport interface, which is implemented in the `FreeRTOS-Plus/Source/Application-Protocols/platform/freertos/transport/src/tls_freertos.c` file. Keep in mind that we're setting the keep-alive seconds for the broker in `xConnectInfo`.

The next function shows how the TLS transport interface and time function are set in an MQTT context using the `MQTT_Init` function. It also shows how an event callback function pointer (`prvEventCallback`) is set. This callback is used for reporting incoming messages.

Subscribing to an MQTT topic

The `prvMQTTSubscribeWithBackoffRetries` function demonstrates how to subscribe to a topic filter on the MQTT broker. The example demonstrates how to subscribe to one topic filter, but it's possible to pass a list of topic filters in the same subscribe API call to subscribe to more than one topic filter. Also, in case the MQTT broker rejects the subscription request, the subscription will retry, with exponential backoff, for `RETRY_MAX_ATTEMPTS`.

Publishing to a topic

The `prvMQTTPublishToTopic` function demonstrates how to publish to a topic on the MQTT broker.

Receiving incoming messages

The application registers an event callback function before it connects to the broker, as described earlier. The `prvMQTTDemoTask` function calls the `MQTT_ProcessLoop` function to receive incoming messages. When an incoming MQTT message is received, it calls the event callback function registered by the application. The `prvEventCallback` function is an example of such an event callback function. `prvEventCallback` examines the incoming packet type and calls the appropriate handler. In the example below, the function either calls `prvMQTTProcessIncomingPublish()` for handling incoming publish messages or `prvMQTTProcessResponse()` to handle acknowledgements (ACK).

Processing incoming MQTT publish packets

The prvMQTTProcessIncomingPublish function demonstrates how to process a publish packet from the MQTT broker.

Unsubscribing from a topic

The last step in the workflow is to unsubscribe from the topic so that the broker won't send any published messages from `mqttexampleTOPIC`. Here is the definition of the function prvMQTTUnsubscribeFromTopic.

Changing the root CA used in the demo

By default, the FreeRTOS demos use the Amazon Root CA 1 certificate (RSA 2048 bit key) to authenticate with the AWS IoT Core server. It is possible to use other CA certificates for server authentication, including the Amazon Root CA 3 certificate (ECC 256 bit key). To change the root CA for the coreMQTT mutual authentication demo:

1. In a text editor, open the *freertos*`/vendors/`*vendor*`/boards/`*board*`/aws_demos/config_files/mqtt_demo_mutual_auth_config.h` file.
2. In the file, locate the following line.

   ```
   * #define democonfigROOT_CA_PEM    "...insert here..."
   ```

   Uncomment this line and, if necessary, move it past the comment block end `*/`.
3. Copy the CA certificate that you want to use and then paste it in the `"...insert here..."` text. The result should look like the following example.

   ```
   #define democonfigROOT_CA_PEM   "-----BEGIN CERTIFICATE-----\n"\
   "MIIBtjCCAVugAwIBAgITBmyf1XSXNmY/Owua2eiedgPySjAKBggqhkjOPQQDAjA5\n"\
   "MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6b24g\n"\
   "Um9vdCBDQSAzMB4XDTE1MDUyNjAwMDAwMFoXDTQwMDUyNjAwMDAwMFowOTELMAkG\n"\
   "A1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJvb3Qg\n"\
   "Q0EgMzBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCmXp8ZBf8ANm+gBG1bG8lKl\n"\
   "ui2yEujSLtf6ycXYqm0fc4E7O5hrOXwzpcVOho6AF2hiRVd9RFgdszflZwjrZt6j\n"\
   "QjBAMA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/BAQDAgGGMB0GA1UdDgQWBBSr\n"\
   "ttvXBp43rDCGB5Fwx5zEGbF4wDAKBggqhkjOPQQDAgNJADBGAiEA4IWSoxe3jfkr\n"\
   "BqWTrBqYaGFy+uGh0PsceGCmQ5nFuMQCIQCcAu/xlJyzlvnrxir4tiz+OpAUFteM\n"\
   "YyRIHN8wfdVoOw==\n"\
   "-----END CERTIFICATE-----\n"
   ```

4. (Optional) You can change the root CA for other demos. Repeat steps 1 through 3 for each *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/*demo-name*_config.h file.

coreMQTT Agent connection sharing demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

The coreMQTT connection sharing demo project shows you how to use a multithreaded application to establish a connection to the AWS MQTT broker using TLS with mutual authentication between the client and the server. This demo uses an mbedTLS-based transport interface implementation to establish a server and client-authenticated TLS connection, and demonstrates the subscribe-publish workflow of MQTT at the QoS 1 level. The demo subscribes to a topic filter, publishes to topics that match the filter, and then waits to receive those messages back from the server at the QoS 1 level. This cycle of publishing to the broker and receiving the same message back from the broker is repeated a number of times for each created task. Messages in this demo are sent at QoS 1, which guarantees at least one delivery according to the MQTT specification.

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

This demo uses a thread safe queue to hold commands to interact with the MQTT API. There are two tasks to take note of in this demo.

- An MQTT Agent (main) task processes the commands from the command queue while other tasks enqueue them. This task enters a loop, during which it processes commands from the command queue. If a termination command is received, this task will break out of the loop.

- A demo subpub task creates a subscription to an MQTT topic, then creates publish operations and pushes them to the command queue. These publish operations are then run by the MQTT Agent task. The demo subpub task waits for the publish to complete, indicated by execution of the command completion callback, then enters a short delay before it starts the next publish. This task shows examples of how application tasks would use the coreMQTT Agent API.

For incoming publish messages, the coreMQTT Agent invokes a single callback function. This demo also includes a subscription manager that allows tasks to specify a callback to invoke for incoming publish messages on topics to which they have subscribed. The agent's incoming publish callback in this demo invokes the subscription manager to fan out publishes to any task that has registered a subscription.

This demo uses a TLS connection with mutual authentication to connect to AWS. If the network unexpectedly disconnects during the demo, then the client attempts to reconnect using exponential backoff logic. If the client successfully reconnects, but the broker can't resume the prior session, then the client will resubscribe to the same topics as the previous session.

Single threaded vs multithreaded

There are two coreMQTT usage models, single threaded and multithreaded (multitasking). The single threaded model uses the coreMQTT library solely from one thread, and requires you to make repeated explicit calls in the MQTT library. Multithreaded use cases can instead run the MQTT protocol in the background within an agent (or daemon) task, as shown in the demo documented here. When you run the MQTT protocol in an agent task you do not have to explicitly manage any MQTT state or call the `MQTT_ProcessLoop` API function. Also, when you use an agent task, multiple application tasks can share a single MQTT connection without the need for synchronization primitives such as mutexes.

Source code

The demo source files are named `mqtt_agent_task.c` and `simple_sub_pub_demo.c` and can be found in the *`freertos`*`/demos/coreMQTT_Agent/` directory and the GitHub website.

Functionality

This demo creates at least two tasks: a primary one that processes requests for MQTT API calls, and a configurable number of subtasks that create those requests. In this demo, the primary task creates the subtasks, calls the processing loop, and cleans up afterwards. The primary task creates a single MQTT connection to the broker that is shared among the subtasks. The subtasks create an

MQTT subscription with the broker and then publish messages to it. Each subtask uses a unique topic for its publishes.

Main task

The main application task, RunCoreMQTTAgentDemo, establishes an MQTT session, creates the subtasks, and runs the processing loop MQTTAgent_CommandLoop until a termination command is received. If the network unexpectedly disconnects, the demo will reconnect to the broker in the background, and re-establish subscriptions with the broker. After the processing loop has terminated, it disconnects from the broker.

Commands

When you invoke a coreMQTT Agent API it creates a command that is sent to the agent task's queue, which is processed in `MQTTAgent_CommandLoop()`. At the time the command is created, optional completion callback and context parameters may be passed. Once the corresponding command is complete, the completion callback will be invoked with the passed context and any return values that were created as a result of the command. The signature for the completion callback is as follows:

```
typedef void (* MQTTAgentCommandCallback_t )( void * pCmdCallbackContext,
                                              MQTTAgentReturnInfo_t * pReturnInfo );
```

The command completion context is user-defined; for this demo, it is: struct MQTTAgentCommandContext.

Commands are considered completed when:

- Subscribes, unsubscribes, and publishes with QoS > 0: Once the corresponding acknowledgment packet has been received.
- All other operations: Once the corresponding coreMQTT API has been invoked.

Any structures used by the command, including publish information, subscription information, and completion contexts, must stay in scope until the command has completed. A calling task must not reuse any of a command's structures before the invocation of the completion callback. Note that since the completion callback is invoked by the MQTT Agent, it will run with the thread context of the agent task, not the task that created the command. Inter-process communication mechanisms, such as task notifications or queues, can be used to signal the calling task of command completion.

Running the command loop

Commands are processed continuously in `MQTTAgent_CommandLoop()`. If there are no commands to be processed, the loop will wait for a maximum of `MQTT_AGENT_MAX_EVENT_QUEUE_WAIT_TIME` for one to be added to the queue, and, if no command is added, it will run a single iteration of `MQTT_ProcessLoop()`. This ensures both that MQTT Keep-Alive is managed, and that any incoming publishes are received even when there are no commands in the queue.

The command loop function will return for the following reasons:

- A command returns any status code besides `MQTTSuccess`. The error status is returned by the command loop, so you may decide how to handle it. In this demo, the TCP connection is reestablished, and a reconnect attempt is made. If there is any error, a reconnection can occur in the background without any intervention from other tasks using MQTT.
- A disconnect command (from `MQTTAgent_Disconnect`) is processed. The command loop exits so that TCP can be disconnected.
- A terminate command (from `MQTTAgent_Terminate`) is processed. This command also marks any command still in the queue or awaiting an acknowledgment packet as an error, with a return code of `MQTTRecvFailed`.

Subscription manager

Since the demo uses multiple topics, a subscription manager is a convenient way to associate subscribed topics with unique callbacks or tasks. The subscription manager in this demo is single-threaded, so it should not be used by multiple tasks concurrently. In this demo, subscription manager functions are only called from callback functions that are passed to the MQTT agent, and run only with the agent task's thread context.

Simple subscribe-publish Task

Each instance of the prvSimpleSubscribePublishTask creates a subscription to an MQTT topic, and creates publish operations for that topic. To demonstrate multiple publish types, even numbered tasks use QoS 0 (which are complete once the publish packet is sent) and odd tasks use QoS 1 (which are complete upon receipt of a PUBACK packet).

Over-the-air updates demo application

FreeRTOS includes a demo application that demonstrates the functionality of the over-the-air (OTA) library. The OTA demo application is located in the *freertos*/demos/ota/ota_demo_core_mqtt/ota_demo_core_mqtt.c or the *freertos*/demos/ota/ota_demo_core_http/ota_demo_core_http.c file.

The OTA demo application does the following:

1. Initializes the FreeRTOS network stack and MQTT buffer pool.
2. Creates a task to exercise the OTA library using `vRunOTAUpdateDemo()`.
3. Creates an MQTT client using `_establishMqttConnection()`.
4. Connects to the AWS IoT MQTT broker using `IotMqtt_Connect()` and registers an MQTT disconnect callback: `prvNetworkDisconnectCallback`.
5. Calls `OTA_AgentInit()` to create the OTA task and registers a callback to be used when the OTA task is complete.
6. Reuses the MQTT connection with `xOTAConnectionCtx.pvControlClient = _mqttConnection;`
7. If MQTT disconnects, the application suspends the OTA agent, tries to reconnect using exponential delay with jitter, and then resumes the OTA agent.

Before you can use OTA updates, complete all prerequisites in FreeRTOS Over-the-Air Updates

After you complete the setup for OTA updates, download, build, flash, and run the FreeRTOS OTA demo on a platform that supports OTA functionality. Device-specific demo instructions are available for the following FreeRTOS-qualified devices:

- Texas Instruments CC3220SF-LAUNCHXL
- Microchip Curiosity PIC32MZEF
- Espressif ESP32
- Download, build, flash and run the FreeRTOS OTA demo on the Renesas RX65N

After you build, flash, and run the OTA demo application on your device, you can use the AWS IoT console or the AWS CLI to create an OTA update job. After you have created an OTA update job, connect a terminal emulator to see the progress of the OTA update. Make a note of any errors generated during the process.

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

```
    249 21207 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    250 21247 [MQTT Agent Task] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
 ReceivedBytes=601.
    251 21247 [MQTT Agent Task] [core_mqtt.c:1045] [INFO] [MQTT] De-serialized incoming
 PUBLISH packet: DeserializerResult=MQTTSuccess.
    252 21248 [MQTT Agent Task] [core_mqtt.c:1058] [INFO] [MQTT] State record updated.
 New state=MQTTPubAckSend.
    253 21249 [MQTT Agent Task] [ota_demo_core_mqtt.c:976] [INFO] [MQTT] Received job
 message callback, size 548.
    254 21252 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobId: AFR_OTA-9702f1a3-b747-4c3e-a0eb-a3b0cf83ddbb]
    255 21253 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.streamname: AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f]
    256 21255 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.protocols: ["MQTT"]]
    257 21256 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[filepath: aws_demos.bin]
    258 21257 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[filesize: 1164016]
    259 21258 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileid: 0]
    260 21259 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[certfile: ecdsa-sha256-signer.crt.pem]
    261 21260 [OTA Agent Task] [ota.c:1575] [INFO] [OTA] Extracted parameter [ sig-
sha256-ecdsa: MEQCIE1SFkIHHiZAvkPpu6McJtx7SYoD... ]
    262 21261 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileType: 0]
    263 21262 [OTA Agent Task] [ota.c:2199] [INFO] [OTA] Job document was accepted.
 Attempting to begin the update.
    264 21263 [OTA Agent Task] [ota.c:2323] [INFO] [OTA] Job parsing success:
 OtaJobParseErr_t=OtaJobParseErrNone, Job name=AFR_OTA-9702f1a3-b747-4c3e-a0eb-
a3b0cf83ddbb
    265 21318 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    266 21418 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    267 21469 [OTA Agent Task] [ota.c:938] [INFO] [OTA] Setting OTA data interface.
```

```
    268 21470 [OTA Agent Task] [ota.c:2839] [INFO] [OTA] Current State=[CreatingFile],
 Event=[ReceivedJobDocument], New state=[CreatingFile]
    269 21482 [MQTT Agent Task] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
 ReceivedBytes=3.
    270 21483 [OTA Agent Task] [ota_demo_core_mqtt.c:1503] [INFO] [MQTT] SUBSCRIBED
 to topic $aws/things/__test_infra_thing71/streams/AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f/data/cbor to bro
    271 21484 [OTA Agent Task] [ota.c:2839] [INFO] [OTA] Current
 State=[RequestingFileBlock], Event=[CreateFile], New state=[RequestingFileBlock]
    272 21518 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    273 21532 [MQTT Agent Task] [core_mqtt_agent_command_functions.c:76] [INFO] [MQTT]
 Publishing message to $aws/things/__test_infra_thing71/streams/AFR_OTA-945d320b-
a18b-441b-b435-4a18d4e7671f/
    274 21534 [OTA Agent Task] [ota_demo_core_mqtt.c:1553] [INFO] [MQTT] Sent PUBLISH
 packet to broker $aws/things/__test_infra_thing71/streams/AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f/get/cbor
    275 21534 [OTA Agent Task] [ota_mqtt.c:1112] [INFO] [OTA] Published to MQTT
 topic to request the next block: topic=$aws/things/__test_infra_thing71/streams/
AFR_OTA-945d320b-a18b-441b-b435-4a1
    276 21537 [OTA Agent Task] [ota.c:2839] [INFO] [OTA] Current
 State=[WaitingForFileBlock], Event=[RequestFileBlock], New state=[WaitingForFileBlock]
    277 21558 [MQTT Agent Task] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
 ReceivedBytes=4217.
    278 21559 [MQTT Agent Task] [core_mqtt.c:1045] [INFO] [MQTT] De-serialized incoming
 PUBLISH packet: DeserializerResult=MQTTSuccess.
    279 21560 [MQTT Agent Task] [core_mqtt.c:1058] [INFO] [MQTT] State record updated.
 New state=MQTTPublishDone.
    280 21561 [MQTT Agent Task] [ota_demo_core_mqtt.c:1026] [INFO] [MQTT] Received data
 message callback, size 4120.
    281 21563 [OTA Agent Task] [ota.c:2464] [INFO] [OTA] Received valid file block:
 Block index=0, Size=4096
    282 21566 [OTA Agent Task] [ota.c:2683] [INFO] [OTA] Number of blocks remaining:
 284

    ... // Output removed for brevity

    3672 42745 [OTA Agent Task] [ota.c:2464] [INFO] [OTA] Received valid file block:
 Block index=284, Size=752
    3673 42747 [OTA Agent Task] [ota.c:2633] [INFO] [OTA] Received final block of the
 update.
    (428298) ota_pal: No such certificate file: ecdsa-sha256-signer.crt.pem. Using
 certificate in ota_demo_config.h.
```

```
   3674 42818 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 285
Queued: 285   Processed: 284   Dropped: 0
   3675 42918 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 285
Queued: 285   Processed: 284   Dropped: 0

   ... // Output removed for brevity

   3678 43197 [OTA Agent Task] [ota.c:2654] [INFO] [OTA] Received entire update and
validated the signature.
   3685 43215 [OTA Agent Task] [ota_demo_core_mqtt.c:862] [INFO] [MQTT] Received
OtaJobEventActivate callback from OTA Agent.

   ... // Output removed for brevity

   2 39 [iot_thread] [INFO ][DEMO][390] ---------STARTING DEMO---------

   [0;32mI (3633) WIFI: WIFI_EVENT_STA_CONNECTED
   [0;32mI (4373) WIFI: SYSTEM_EVENT_STA_GOT_IP

   ... // Output removed for brevity

   4 351 [sys_evt] [INFO ][DEMO][3510] Connected to WiFi access point, ip address:
255.255.255.0.
   5 351 [iot_thread] [INFO ][DEMO][3510] Successfully initialized the demo. Network
type for the demo: 1
   6 351 [iot_thread] [ota_demo_core_mqtt.c:1902] [INFO] [MQTT] OTA over MQTT demo,
Application version 0.9.1
   7 351 [iot_thread] [ota_demo_core_mqtt.c:1323] [INFO] [MQTT] Creating a TLS
connection to <endpoint>-ats.iot.us-west-2.amazonaws.com:8883.
   9 718 [iot_thread] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
ReceivedBytes=2.
   10 718 [iot_thread] [core_mqtt_serializer.c:970] [INFO] [MQTT] CONNACK session
present bit not set.
   11 718 [iot_thread] [core_mqtt_serializer.c:912] [INFO] [MQTT] Connection accepted.

   ... // Output removed for brevity

   17 736 [OTA Agent Task] [ota_demo_core_mqtt.c:1503] [INFO] [MQTT] SUBSCRIBED to
topic $aws/things/__test_infra_thing71/jobs/notify-next to broker.
   18 737 [OTA Agent Task] [ota_mqtt.c:381] [INFO] [OTA] Subscribed to MQTT topic:
$aws/things/__test_infra_thing71/jobs/notify-next
   30 818 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
Queued: 0   Processed: 0   Dropped: 0
```

```
    31 819 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobId: AFR_OTA-9702f1a3-b747-4c3e-a0eb-a3b0cf83ddbb]
    32 820 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.statusDetails.updatedBy: 589824]
    33 822 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.streamname: AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f]
    34 823 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.protocols: ["MQTT"]]
    35 824 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[filepath: aws_demos.bin]
    36 825 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[filesize: 1164016]
    37 826 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileid: 0]
    38 827 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[certfile: ecdsa-sha256-signer.crt.pem]
    39 828 [OTA Agent Task] [ota.c:1575] [INFO] [OTA] Extracted parameter [ sig-sha256-
ecdsa: MEQCIE1SFkIHHiZAvkPpu6McJtx7SYoD... ]
    40 829 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileType: 0]
    41 830 [OTA Agent Task] [ota.c:2102] [INFO] [OTA] In self test mode.
    42 830 [OTA Agent Task] [ota.c:1936] [INFO] [OTA] New image has a higher version
 number than the current image: New image version=0.9.1, Previous image version=0.9.0
    43 832 [OTA Agent Task] [ota.c:2120] [INFO] [OTA] Image version is valid: Begin
 testing file: File ID=0
    53 896 [OTA Agent Task] [ota.c:794] [INFO] [OTA] Beginning self-test.
    62 971 [OTA Agent Task] [ota_demo_core_mqtt.c:1553] [INFO] [MQTT] Sent PUBLISH
 packet to broker $aws/things/__test_infra_thing71/jobs/AFR_OTA-9702f1a3-b747-4c3e-
a0eb-a3b0cf83ddbb/update to br63 971 [MQTT Agent Task] [core_mqtt.c:1045] [INFO] [MQTT]
 De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
    65 973 [MQTT Agent Task] [core_mqtt.c:1058] [INFO] [MQTT] State record updated. New
 state=MQTTPublishDone.
    64 973 [OTA Agent Task] [ota_demo_core_mqtt.c:902] [INFO] [MQTT] Successfully
 updated with the new image.
```

Over-the-air demo configurations

The OTA demo configurations are demo-specific configuration options provided in `aws_iot_ota_update_demo.c`. These configurations are different from the OTA library configurations provided in the OTA library config file.

OTA_DEMO_KEEP_ALIVE_SECONDS

For the MQTT client, this configuration is the maximum time interval that can elapse between finishing the transmission of one control packet and starting to send the next. In the absence of a control packet, a PINGREQ is sent. The broker must disconnect a client that doesn't send a message or a PINGREQ packet in one and a half times of this keep alive interval. This configuration should be adjusted based on the application's requirements.

OTA_DEMO_CONN_RETRY_BASE_INTERVAL_SECONDS

The base interval, in seconds, before retrying the network connection. The OTA demo will try to reconnect after this base time interval. The interval is doubled after every failed attempt. A random delay, up to a maximum of this base delay, is also added to the interval.

OTA_DEMO_CONN_RETRY_MAX_INTERVAL_SECONDS

The maximum interval, in seconds, before retrying the network connection. The reconnect delay is doubled on every failed attempt, but it can go only up to this maximum value, plus a jitter of the same interval.

Download, build, flash, and run the FreeRTOS OTA demo on the Texas Instruments CC3220SF-LAUNCHXL

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

To download FreeRTOS and the OTA demo code

- You can download the source code on the GitHub site at https://github.com/FreeRTOS/FreeRTOS.

To build the demo application

1. Follow the instructions in Get Started with FreeRTOS to import the `aws_demos` project into Code Composer Studio, configure your AWS IoT endpoint, your Wi-Fi SSID and password, and a private key and certificate for your board.
2. Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED`, and define `CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED` or `CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED`.
3. Build the solution and make sure it builds without errors.
4. Start a terminal emulator and use the following settings to connect to your board:
   - Baud rate: 115200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
5. Run the project on your board to confirm it can connect to Wi-Fi and the AWS IoT MQTT message broker.

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

```
0 1000 [Tmr Svc] Simple Link task created
Device came up in Station mode
1 2534 [Tmr Svc] Write certificate...
2 5486 [Tmr Svc] [ERROR] Failed to destroy object. PKCS11_PAL_DestroyObject failed.
3 5486 [Tmr Svc] Write certificate...
4 5776 [Tmr Svc] Security alert threshold = 15
5 5776 [Tmr Svc] Current number of alerts = 1
6 5778 [Tmr Svc] Running Demos.
7 5779 [iot_thread] [INFO ][DEMO][5779] ---------STARTING DEMO---------
8 5779 [iot_thread] [INFO ][INIT][5779] SDK successfully initialized.
Device came up in Station mode
[WLAN EVENT] STA Connected to the AP: afrlab-pepper , BSSID: 74:83:c2:b4:46:27
[NETAPP EVENT] IP acquired by the device
Device has connected to afrlab-pepper
Device IP Address is 192.168.36.176
```

```
    9 8283 [iot_thread] [INFO ][DEMO][8282] Successfully initialized the demo. Network
 type for the demo: 1
    10 8283 [iot_thread] [INFO] OTA over MQTT demo, Application version 0.9.0
    11 8283 [iot_thread] [INFO] Creating a TLS connection to <endpoint>-ats.iot.us-
west-2.amazonaws.com:8883.
    12 8852 [iot_thread] [INFO] Creating an MQTT connection to <endpoint>-ats.iot.us-
west-2.amazonaws.com.
    13 8914 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
    14 8914 [iot_thread] [INFO] CONNACK session present bit not set.
    15 8914 [iot_thread] [INFO] Connection accepted.
    16 8914 [iot_thread] [INFO] Received MQTT CONNACK successfully from broker.
    17 8914 [iot_thread] [INFO] MQTT connection established with the broker.
    18 8915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
    19 8953 [OTA Agent T] [INFO] Current State=[RequestingJob], Event=[Start], New
 state=[RequestingJob]
    20 9008 [MQTT Agent ] [INFO] Packet received. ReceivedBytes=3.
    21 9015 [OTA Agent T] [INFO] SUBSCRIBED to topic $aws/things/__test_infra_thing73/
jobs/notify-next to broker.
    22 9015 [OTA Agent T] [INFO] Subscribed to MQTT topic: $aws/things/
__test_infra_thing73/jobs/notify-next
    23 9504 [MQTT Agent ] [INFO] Publishing message to $aws/things/
__test_infra_thing73/jobs/$next/get.
    24 9535 [MQTT Agent ] [INFO] Packet received. ReceivedBytes=2.
    25 9535 [MQTT Agent ] [INFO] Ack packet deserialized with result: MQTTSuccess.
    26 9536 [MQTT Agent ] [INFO] State record updated. New state=MQTTPublishDone.
    27 9537 [OTA Agent T] [INFO] Sent PUBLISH packet to broker $aws/things/
__test_infra_thing73/jobs/$next/get to broker.
    28 9537 [OTA Agent T] [WARN] OTA Timer handle NULL for Timerid=0, can't stop.
    29 9537 [OTA Agent T] [INFO] Current State=[WaitingForJob],
 Event=[RequestJobDocument], New state=[WaitingForJob]
    30 9539 [MQTT Agent ] [INFO] Packet received. ReceivedBytes=120.
    31 9539 [MQTT Agent ] [INFO] De-serialized incoming PUBLISH packet:
 DeserializerResult=MQTTSuccess.
    32 9540 [MQTT Agent ] [INFO] State record updated. New state=MQTTPublishDone.
    33 9540 [MQTT Agent ] [INFO] Received job message callback, size 62.
    34 9616 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution
    35 9616 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobId
    36 9617 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobDocument
    37 9617 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobDocument.afr_ota
```

```
   38 9617 [OTA Agent T] [INFO] Failed job document content
check: Required job document parameter was not extracted:
parameter=execution.jobDocument.afr_ota.protocols
   39 9618 [OTA Agent T] [INFO] Failed job document content check: Required job
document parameter was not extracted: parameter=execution.jobDocument.afr_ota.files
   40 9618 [OTA Agent T] [INFO] Failed job document content check: Required job
document parameter was not extracted: parameter=filesize
   41 9618 [OTA Agent T] [INFO] Failed job document content check: Required job
document parameter was not extracted: parameter=fileid
   42 9619 [OTA Agent T] [INFO] Failed to parse JSON document as AFR_OTA job:
DocParseErr_t=7
   43 9619 [OTA Agent T] [INFO] No active job available in received job document:
OtaJobParseErr_t=OtaJobParseErrNoActiveJobs
   44 9619 [OTA Agent T] [ERROR] Failed to execute state transition handler: Handler
returned error: OtaErr_t=OtaErrJobParserError
   45 9620 [OTA Agent T] [INFO] Current State=[WaitingForJob],
Event=[ReceivedJobDocument], New state=[CreatingFile]
   46 9915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   47 10915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   48 11915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   49 12915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   50 13915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   51 14915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
```

Download, build, flash, and run the FreeRTOS OTA demo on the Microchip Curiosity PIC32MZEF

Important

This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Note

In agreement with Microchip, we are removing the Curiosity PIC32MZEF (DM320104) from the FreeRTOS Reference Integration repository main branch and will no longer carry it in new releases. Microchip has issued an official notice that the PIC32MZEF (DM320104) is no longer recommended for new designs. The PIC32MZEF projects and source code can still be accessed through the previous release tags. Microchip recommends that customers

use the Curiosity PIC32MZ-EF-2.0 Development board (DM320209) for new designs. The PIC32MZv1 platform can still be found in v202012.00 of the FreeRTOS Reference Integration repository. However, the platform is no longer supported by v202107.00 of the FreeRTOS Reference.

To download the FreeRTOS OTA demo code

- You can download the source code on the GitHub site at https://github.com/FreeRTOS/FreeRTOS.

To build the OTA update demo application

1. Follow the instructions in Get Started with FreeRTOS to import the `aws_demos` project into the MPLAB X IDE, configure your AWS IoT endpoint, your Wi-Fi SSID and password, and a private key and certificate for your board.
2. Open the `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/ota_demo_config.h` file, and enter your certificate.

   ```
   [] = "your-certificate-key";
   ```

3. Paste the contents of your code-signing certificate here:

   ```
   #define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
   ```

   Follow the same format as `aws_clientcredential_keys.h` -- each line must end with the new line character ('\n') and be enclosed in quotation marks.

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

   ```
   "-----BEGIN CERTIFICATE-----\n"
   "MIIBXTCCAQOgAwIBAgIJAM4DeybZcTwKMAoGCCqGSM49BAMCMCExHzAdBgNVBAMM\n"
   "FnRlc3Rf62lnbmVyQGFtYXpvbi5jb20wHhcNMTcxMTAzMTkxODM1WhcNMTgxMTAz\n"
   "MTkxODM2WjAhMR8wHQYDVQBBZZZ0ZXN0X3NpZ25lckBhbWF6b24uY29tMFkwEwYH\n"
   "KoZIzj0CAQYIKoZIzj0DAQcDQgAERavZfvwL1X+E4dIF7dbkVMUn4IrJ1CAsFkc8\n"
   "gZxPzn683H40XMKltDZPEwr9ng78w9+QYQg7ygnr2stz8yhh06MkMCIwCwYDVR0P\n"
   "BAQDAgeAMBMGA1UdJQQMMAoGCCsGAQUFBwMDMAoGCCqGSM49BAMCA0gAMEUCIF0R\n"
   "r5cb7rEUNtWOvGd05MacrgOABfSoVYvBOK9fP63WAqt5h3BaS123coKSGg84twlq\n"
   "TkO/pV/xEmyZmZdV+HxV/OM=\n"
   ```

```
"-----END CERTIFICATE-----\n";
```

4. Install Python 3 or later.
5. Install pyOpenSSL by running `pip install pyopenssl`.
6. Copy your code-signing certificate in .pem format in the path `demos/ota/bootloader/utility/codesigner_cert_utility/`. Rename the certificate file `aws_ota_codesigner_certificate.pem`.
7. Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED`, and define `CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED` or `CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED`.
8. Build the solution and make sure it builds without errors.
9. Start a terminal emulator and use the following settings to connect to your board:
   - Baud rate: 115200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
10. Unplug the debugger from your board and run the project on your board to confirm it can connect to Wi-Fi and the AWS IoT MQTT message broker.

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

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

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

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

```
[prvBOOT_ValidateImages] Booting default image.


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

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


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

```
33 44964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
34 45964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
35 46964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
36 47964 [OTA] State: Ready  Received: 1   Queued: 1   Processed: 1   Dropped: 0
```

The terminal emulator should display text like the following:

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


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

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

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

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

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

Firmware ver. : 4.2.1

Min driver ver : 4.2.1

Curr driver ver: 4.2.1

WILC1000: Initialization successful!

Start Wi-Fi Connection...
Wi-Fi Connected
2 7219 [IP-task] vDHCPProcess: offer c0a804beip
3 7230 [IP-task] vDHCPProcess: offer c0a804beip
4 7230 [IP-task]

IP Address: 192.168.4.190
5 7230 [IP-task] Subnet Mask: 255.255.240.0
6 7230 [IP-task] Gateway Address: 192.168.0.1
```

```
7 7230 [IP-task] DNS Server Address: 208.67.222.222


8 7232 [OTA] OTA demo version 0.9.0
9 7232 [OTA] Creating MQTT Client...
10 7232 [OTA] Connecting to broker...
11 7232 [OTA] Sending command to MQTT task.
12 7232 [MQTT] Received message 10000 from queue.
13 8501 [IP-task] Socket sending wakeup to MQTT task.
14 10207 [MQTT] Received message 0 from queue.
15 10256 [IP-task] Socket sending wakeup to MQTT task.
16 10256 [MQTT] Received message 0 from queue.
17 10256 [MQTT] MQTT Connect was accepted. Connection established.
18 10256 [MQTT] Notifying task.
19 10257 [OTA] Command sent to MQTT task passed.
20 10257 [OTA] Connected to broker.
21 10258 [OTA Task] Sending command to MQTT task.
22 10258 [MQTT] Received message 20000 from queue.
23 10306 [IP-task] Socket sending wakeup to MQTT task.
24 10306 [MQTT] Received message 0 from queue.
25 10306 [MQTT] MQTT Subscribe was accepted. Subscribed.
26 10306 [MQTT] Notifying task.
27 10307 [OTA Task] Command sent to MQTT task passed.
28 10307 [OTA Task] [OTA] Subscribed to topic: $aws/things/Microchip/jobs/$next/get/
accepted

29 10307 [OTA Task] Sending command to MQTT task.
30 10307 [MQTT] Received message 30000 from queue.
31 10336 [IP-task] Socket sending wakeup to MQTT task.
32 10336 [MQTT] Received message 0 from queue.
33 10336 [MQTT] MQTT Subscribe was accepted. Subscribed.
34 10336 [MQTT] Notifying task.
35 10336 [OTA Task] Command sent to MQTT task passed.
36 10336 [OTA Task] [OTA] Subscribed to topic: $aws/things/Microchip/jobs/notify-next

37 10336 [OTA Task] [OTA] Check For Update #0
38 10336 [OTA Task] Sending command to MQTT task.
39 10336 [MQTT] Received message 40000 from queue.
40 10366 [IP-task] Socket sending wakeup to MQTT task.
41 10366 [MQTT] Received message 0 from queue.
42 10366 [MQTT] MQTT Publish was successful.
43 10366 [MQTT] Notifying task.
44 10366 [OTA Task] Command sent to MQTT task passed.
45 10376 [IP-task] Socket sending wakeup to MQTT task.
```

```
46 10376 [MQTT] Received message 0 from queue.
47 10376 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:Microchip ]
48 10376 [OTA Task] [OTA] Missing job parameter: execution
49 10376 [OTA Task] [OTA] Missing job parameter: jobId
50 10376 [OTA Task] [OTA] Missing job parameter: jobDocument
51 10378 [OTA Task] [OTA] Missing job parameter: ts_ota
52 10378 [OTA Task] [OTA] Missing job parameter: files
53 10378 [OTA Task] [OTA] Missing job parameter: streamname
54 10378 [OTA Task] [OTA] Missing job parameter: certfile
55 10378 [OTA Task] [OTA] Missing job parameter: filepath
56 10378 [OTA Task] [OTA] Missing job parameter: filesize
57 10378 [OTA Task] [OTA] Missing job parameter: sig-sha256-ecdsa
58 10378 [OTA Task] [OTA] Missing job parameter: fileid
59 10378 [OTA Task] [OTA] Missing job parameter: attr
60 10378 [OTA Task] [OTA] Returned buffer to MQTT Client.
61 11367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
62 12367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
63 13367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
64 14367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
65 15367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
66 16367 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
```

This output shows the Microchip Curiosity PIC32MZEF can connect to AWS IoT and subscribe to the MQTT topics required for OTA updates. The `Missing job parameter` messages are expected because there are no OTA update jobs pending.

Download, build, flash, and run the FreeRTOS OTA demo on the Espressif ESP32

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

1. Download the FreeRTOS source from GitHub. See the README.md file for instructions. Create a project in your IDE that includes all required sources and libraries.
2. Follow the instructions in Getting Started with Espressif to set up the required GCC-based toolchain.

3. Open *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h, comment out #define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED, and define CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED or CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED.
4. Build the demo project by running make in the vendors/espressif/boards/esp32/aws_demos directory. You can flash the demo program and verify its output by running make flash monitor, as described in Getting Started with Espressif.
5. Before running the OTA Update demo:

   - Open *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h, comment out #define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED, and define CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED or CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED.
   - Open vendors/*vendor*/boards/*board*/aws_demos/config_files/ota_demo_config.h, and copy your SHA-256/ECDSA code-signing certificate in:

     ```
     #define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
     ```

Download, build, flash and run the FreeRTOS OTA demo on the Renesas RX65N

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This chapter shows you how download, build, flash and run the FreeRTOS OTA demo applications on the Renesas RX65N.

Topics

- Set up your operating environment
- Set up your AWS resources

- Import, configure the header file and build aws_demos and boot_loader

Set up your operating environment

The procedures in this section use the following environments:

- **IDE**: e^2 studio 7.8.0, e^2 studio 2020-07
- **Toolchains**: CCRX Compiler v3.0.1
- **Target devices**: RSKRX65N-2MB
- **Debuggers**: E^2, E^2 Lite emulator
- **Software**: Renesas Flash Programmer, Renesas Secure Flash Programmer.exe, Tera Term

To set up your hardware

1. Connect the E^2 Lite emulator and USB serial port to your RX65N board and PC.
2. Connect the power source to the RX65N.

Set up your AWS resources

1. To run the FreeRTOS demos, you must have an AWS account with an IAM user that has permission to access AWS IoT services. If you haven't already, follow the steps in Setting up your AWS account and permissions.
2. To set up for OTA updates, follow the steps in OTA update prerequisites. In particular, follow the steps in Prerequisites for OTA updates using MQTT.
3. Open the AWS IoT console.
4. In the left navigation pane, choose **Manage**, then choose **Things** to create a thing.

   A thing is a representation of a device or logical entity in AWS IoT. It can be a physical device or sensor (for example, a light bulb or a switch on a wall). It can also be a logical entity like an instance of an application or physical entity that doesn't connect to AWS IoT, but is related to devices that do (for example, a car that has engine sensors or a control panel). AWS IoT provides a thing registry that helps you manage your things.

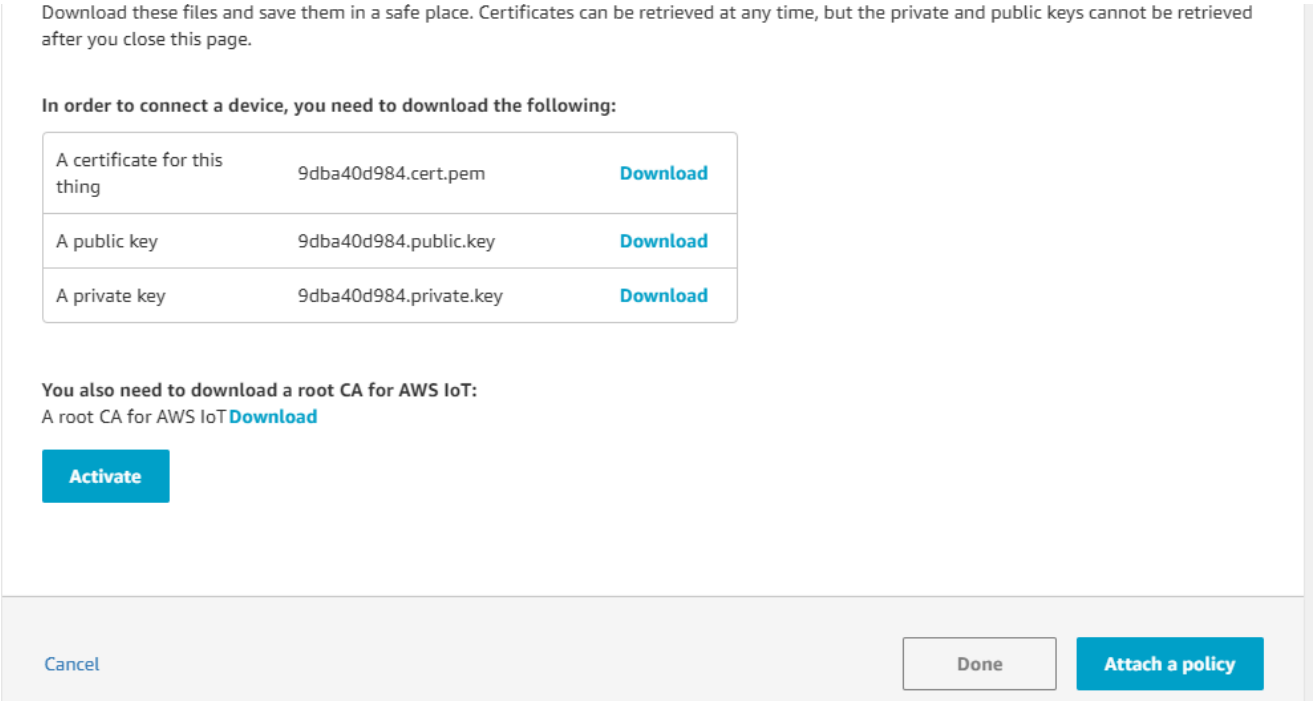
   a. Choose **Create**, then choose **Create a single thing**.

   b. Enter a **Name** for your thing, then choose **Next**.

   c. Choose **Create certificate**.

d. Download the three files that are created and then choose **Activate**.

e. Choose **Attach a policy**.



f. Select the policy that you created in Device policy.

Each device that receives an OTA update using MQTT must be registered as a thing in AWS IoT and the thing must have an attached policy like the one listed. You can find more information about the items in the `"Action"` and `"Resource"` objects at AWS IoT Core Policy Actions and AWS IoT Core Action Resources.

Notes

- The `iot:Connect` permissions allow your device to connect to AWS IoT over MQTT.
- The `iot:Subscribe` and `iot:Publish` permissions on the topics of AWS IoT jobs (`.../jobs/*`) allow the connected device to receive job notifications and job documents, and to publish the completion state of a job execution.
- The `iot:Subscribe` and `iot:Publish` permissions on the topics of AWS IoT OTA streams (`.../streams/*`) allow the connected device to fetch OTA update data from AWS IoT. These permissions are required to perform firmware updates over MQTT.
- The `iot:Receive` permissions allow AWS IoT Core to publish messages on those topics to the connected device. This permission is checked on every delivery of an MQTT

message. You can use this permission to revoke access to clients that are currently subscribed to a topic.

5. To create a code-signing profile and register a code-signing certificate on AWS.

   a. To create the keys and certification, see section 7.3 "Generating ECDSA-SHA256 Key Pairs with OpenSSL" in Renesas MCU Firmware Update Design Policy.

   b. Open the AWS IoT console. In the left navigation pane, select **Manage**, then **Jobs**. Select **Create a job** then **Create OTA update Job**.

   c. Under **Select devices to update** choose **Select** then choose the thing you created previously. Select **Next**.

   d. On the **Create a FreeRTOS OTA update job** page, do the following:

      i. For **Select the protocol for firmware image transfer**, choose **MQTT**.

      ii. For **Select and sign your firmware image**, choose **Sign a new firmware image for me**.

      iii. For **Code signing profile**, choose **Create**.

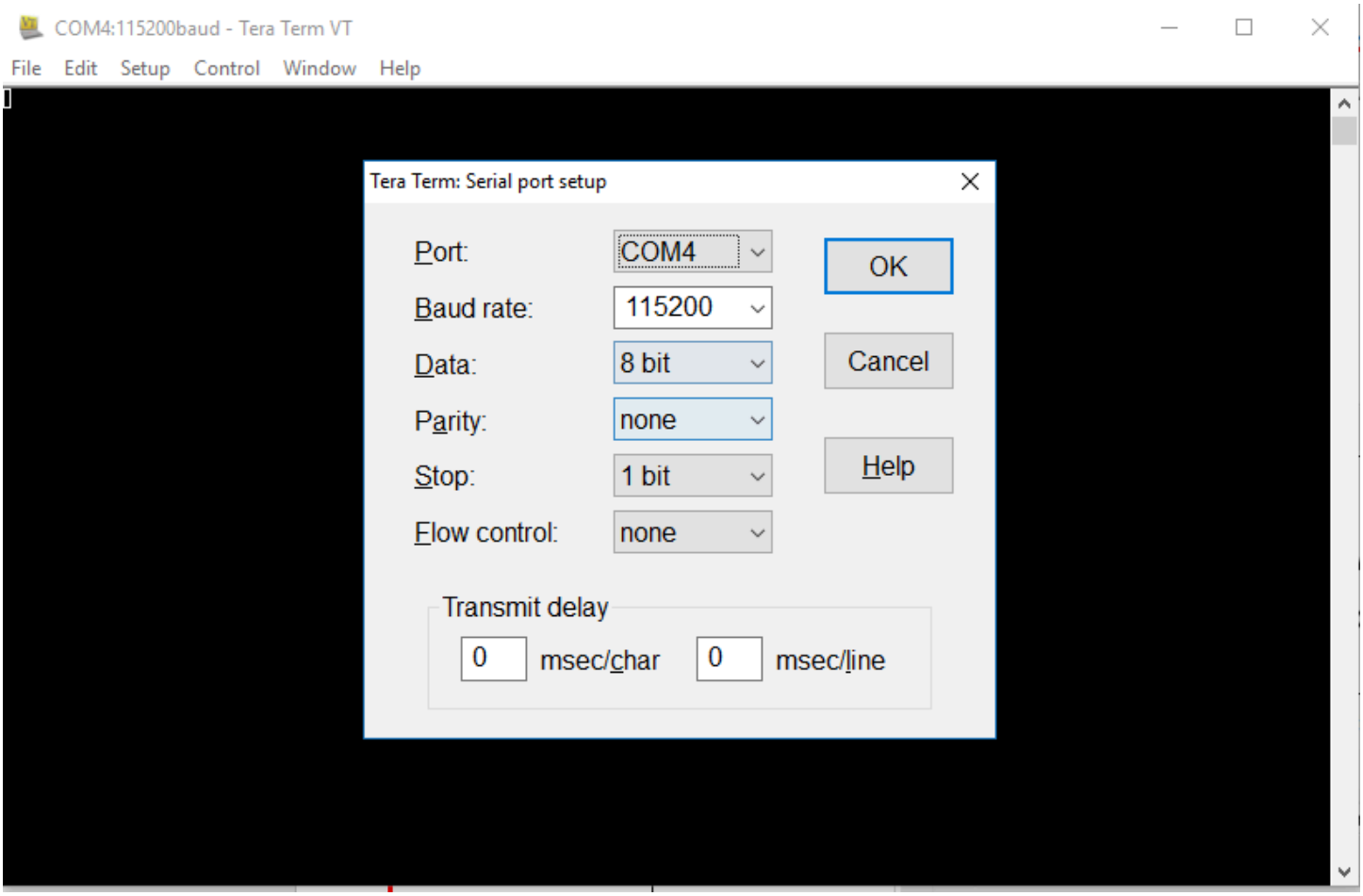
      iv. In the **Create a code signing profile** window, enter a **Profile name**. For the **Device hardware platform** select **Windows Simulator**. For the **Code signing certificate** choose **Import**.

      v. Browse to select the certificate (`secp256r1.crt`), the certificate private key (`secp256r1.key`), and the certificate chain (`ca.crt`).

      vi. Enter a **Pathname of code signing certificate on device**. Then choose **Create**.

6. To grant access to code signing for AWS IoT, follow the steps in Grant access to code signing for AWS IoT.

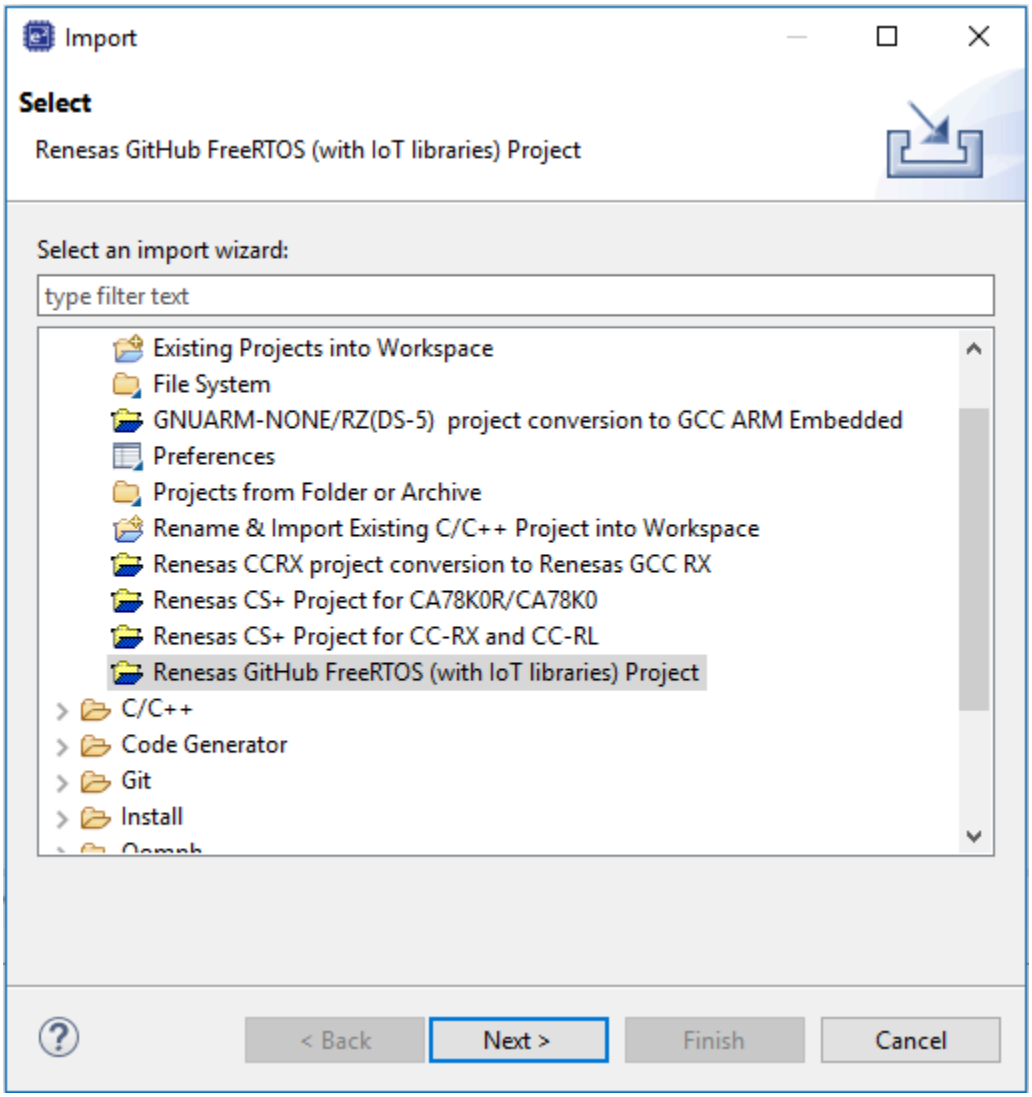
If don't have Tera Term installed on your PC, you can download it from https://ttssh2.osdn.jp/index.html.en and set it up as shown here. Make sure that you plug in the USB Serial port from your device to your PC.



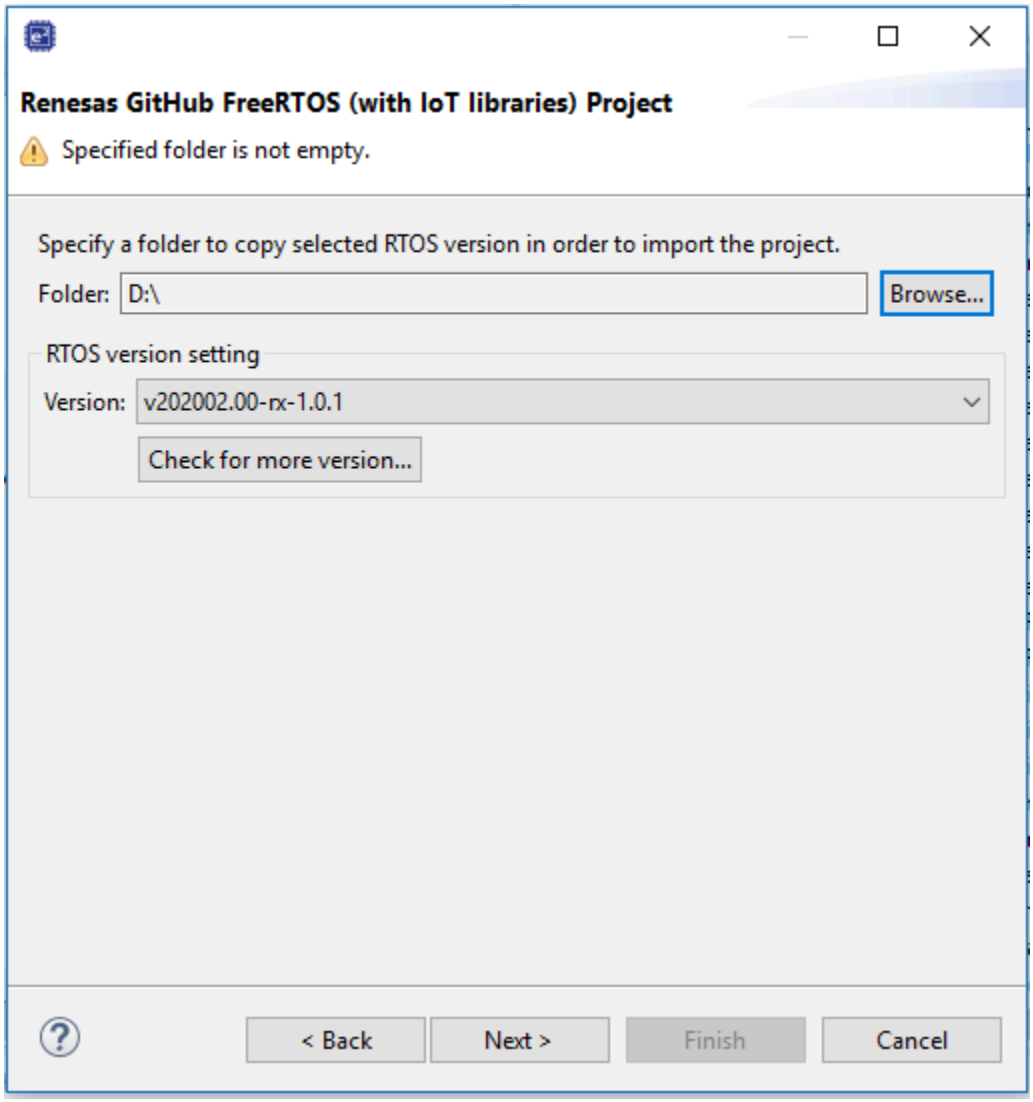
Import, configure the header file and build aws_demos and boot_loader

To begin, you select the latest version of the FreeRTOS package, and this will be downloaded from GitHub and imported automatically into the project. This way you can focus on the configuring FreeRTOS and writing application code.

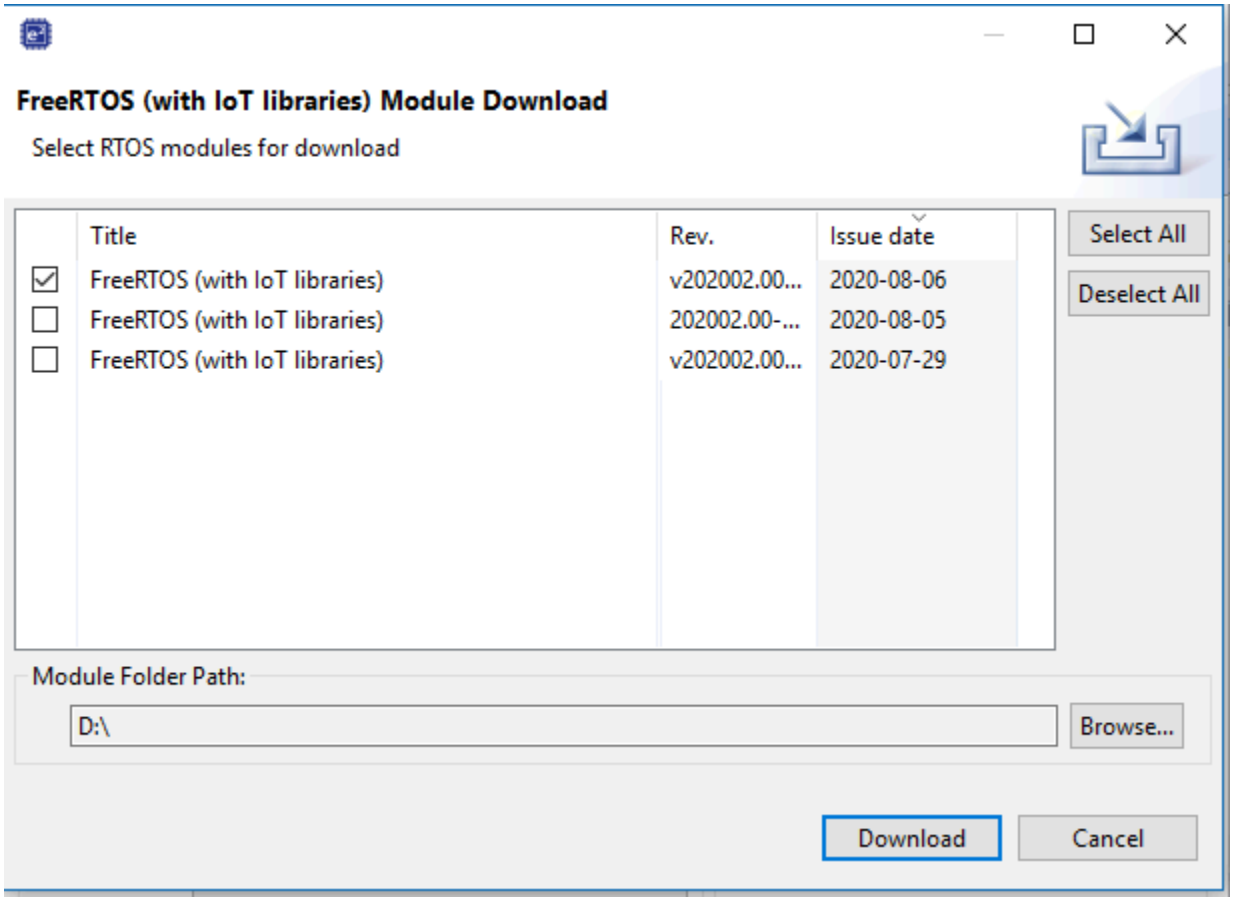
1. Launch e^2 studio.
2. Choose **File**, and then choose **Import...**.
3. Select **Renesas GitHub FreeRTOS (with IoT libraries) Project**.



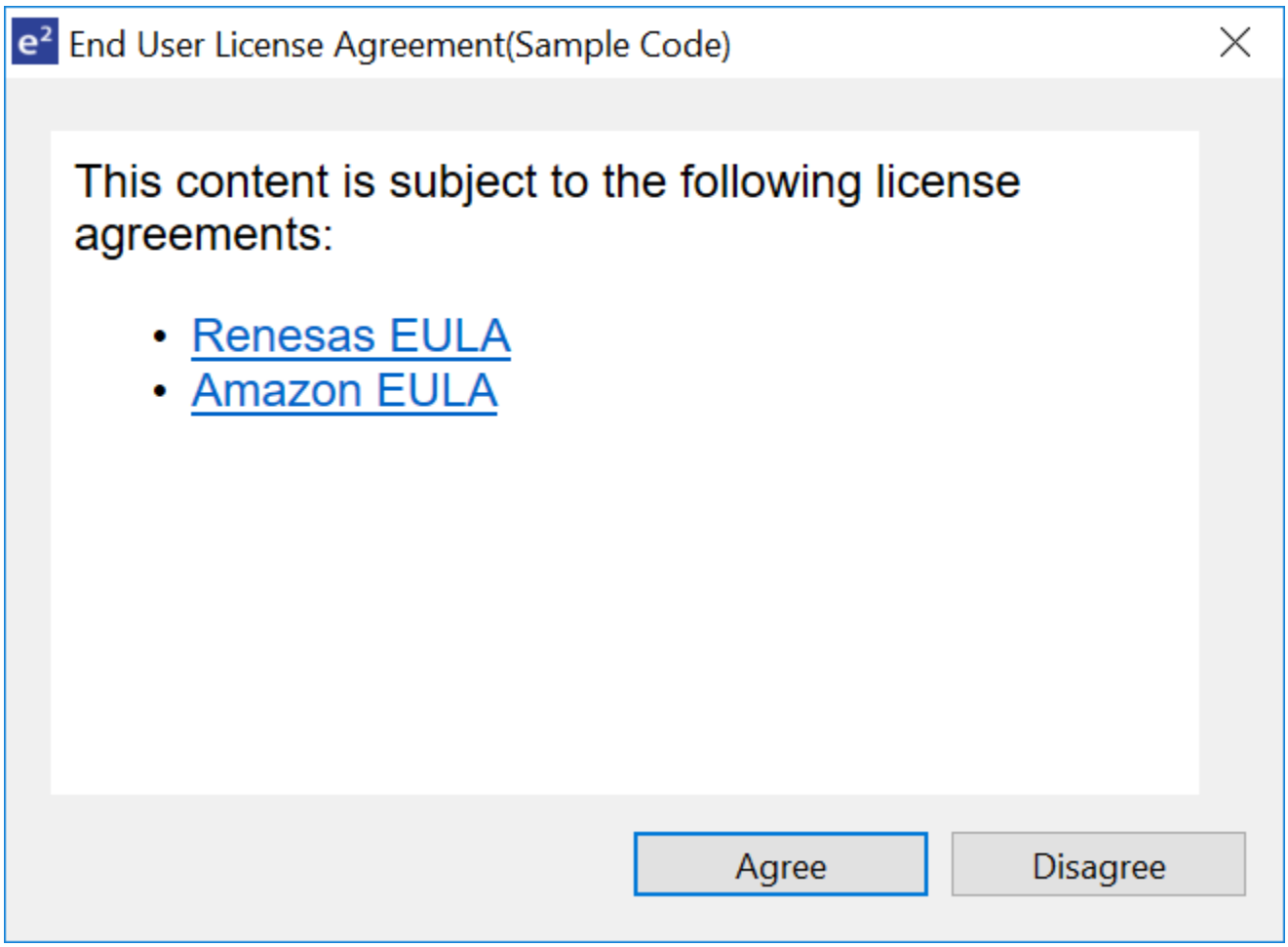
4. Choose **Check for more version...** to show the download dialog box.



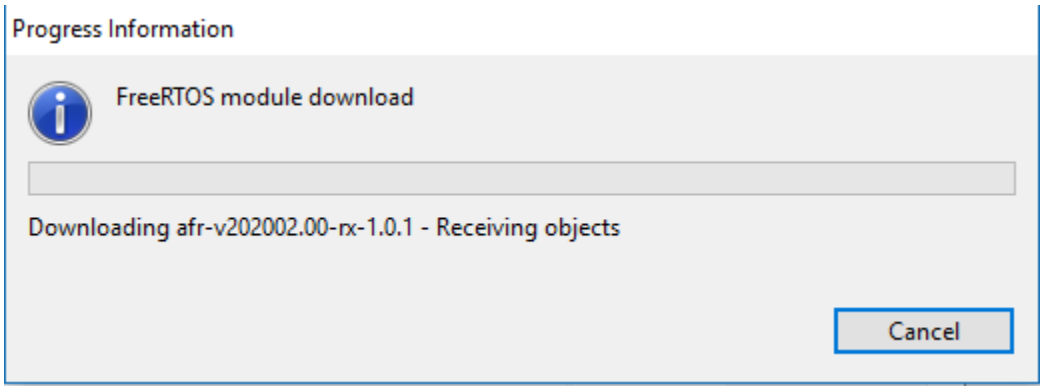
5. Select the latest package.



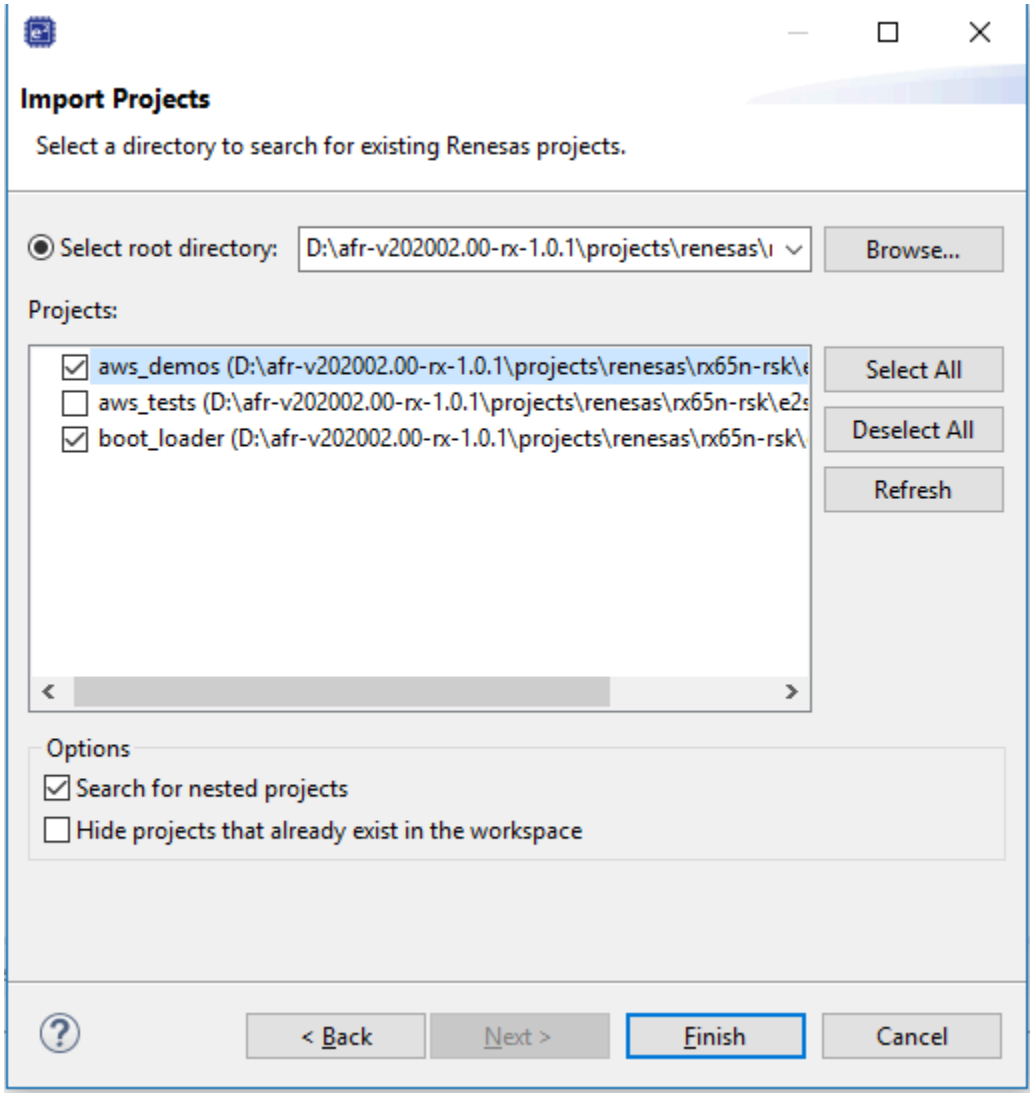
6. Choose **Agree** to accept the end user license agreement.



7. Wait for the download to complete.



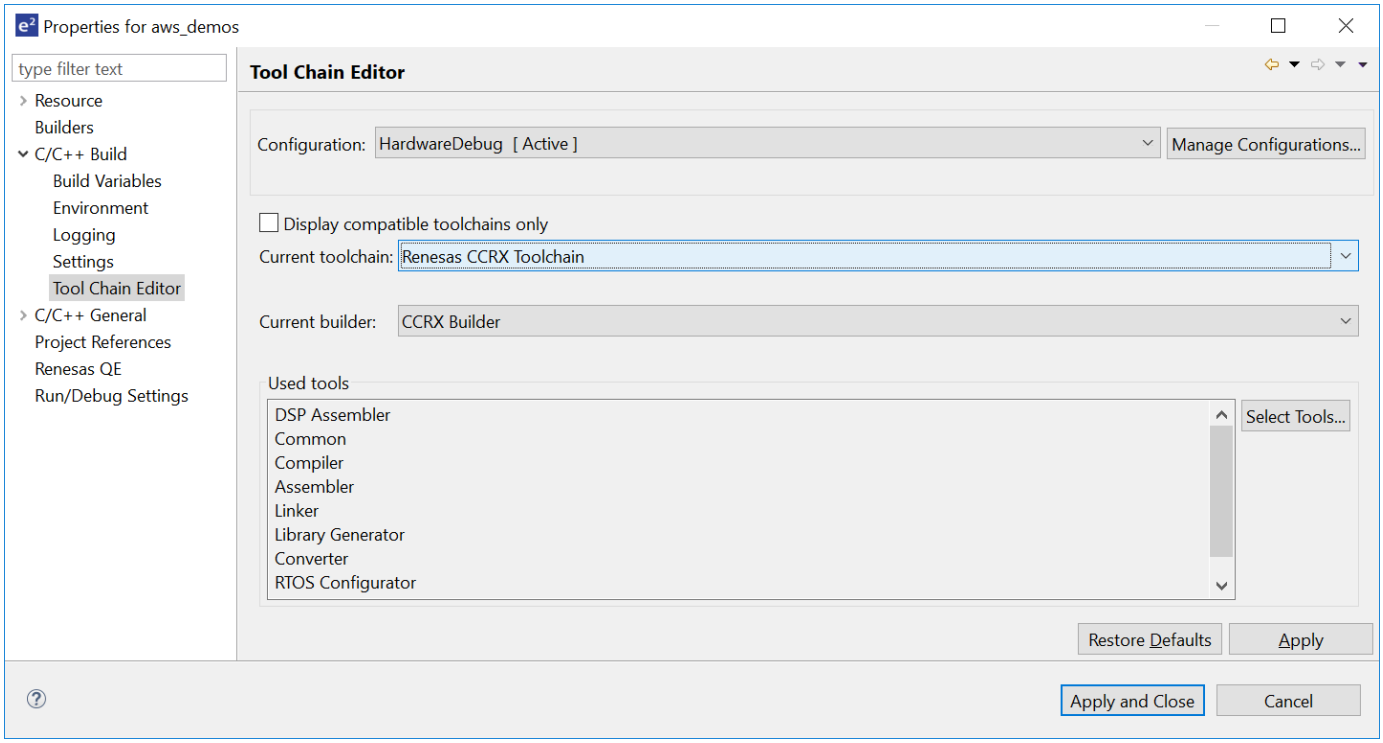
8. Select the **aws_demos** and **boot_loader** projects, then choose **Finish** to import them.



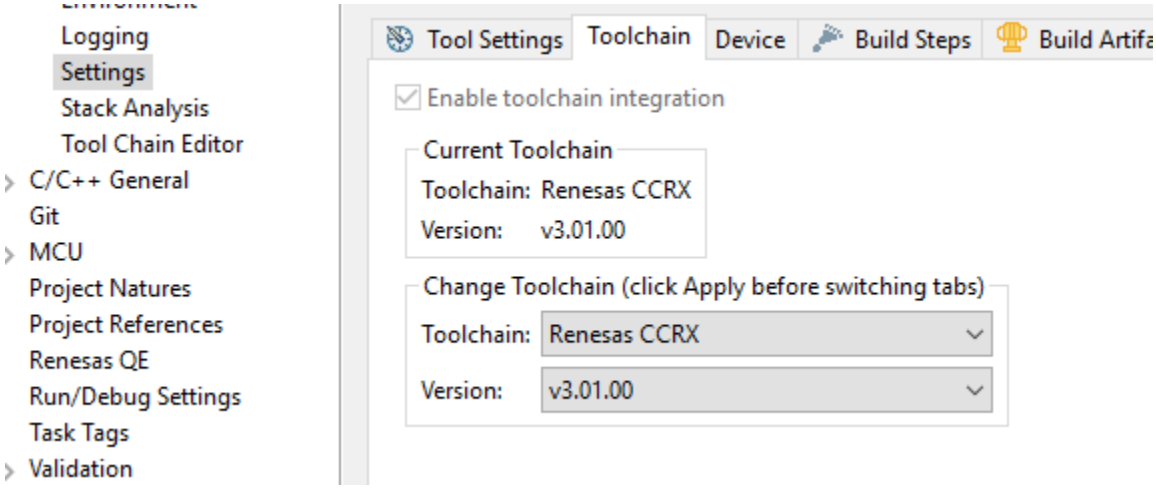
9. For both projects, open the project properties. In the navigation pane, choose **Tool Chain Editor**.

   a. Choose the **Current toolchain**.

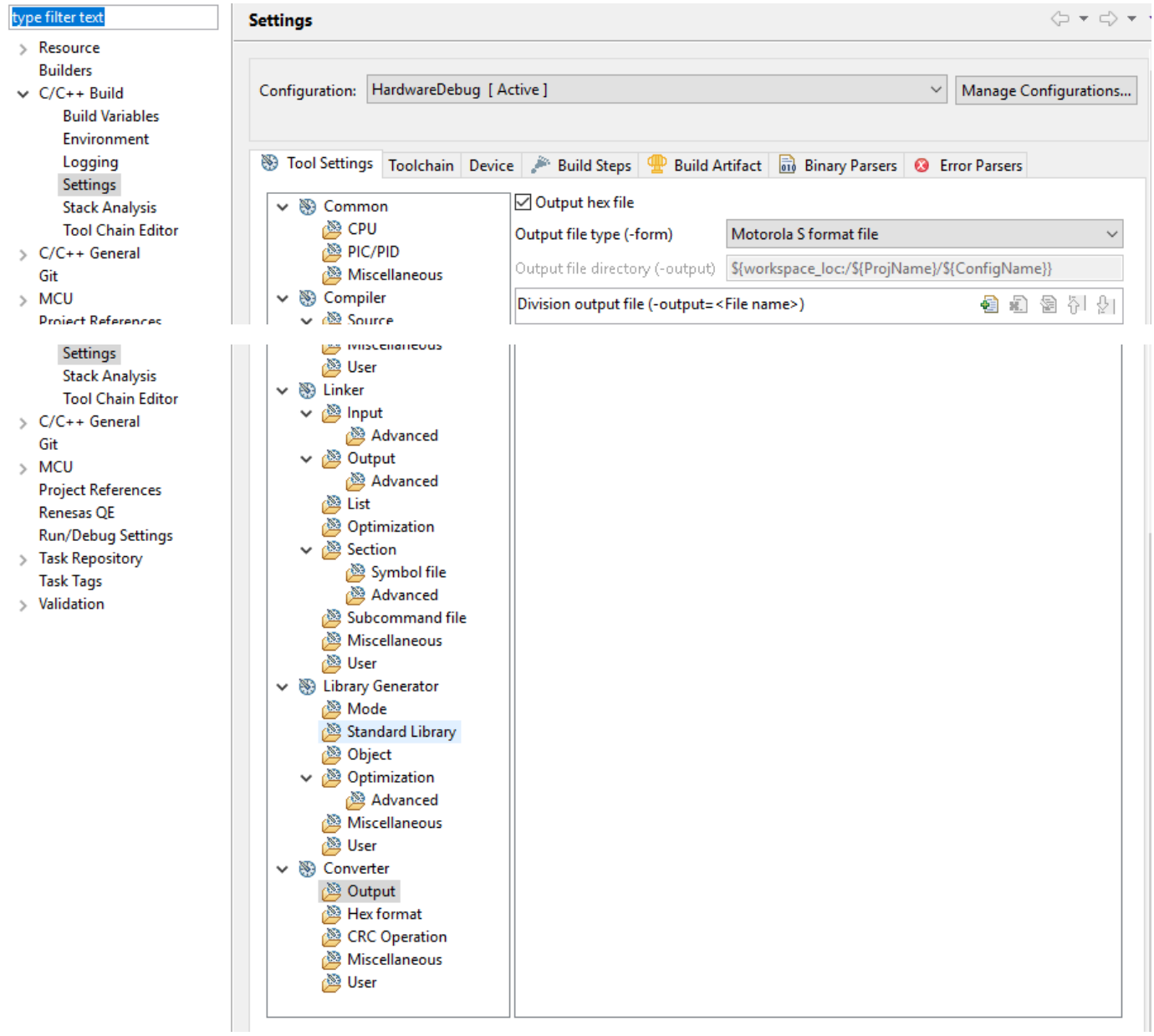
   b. Choose the **Current builder**.



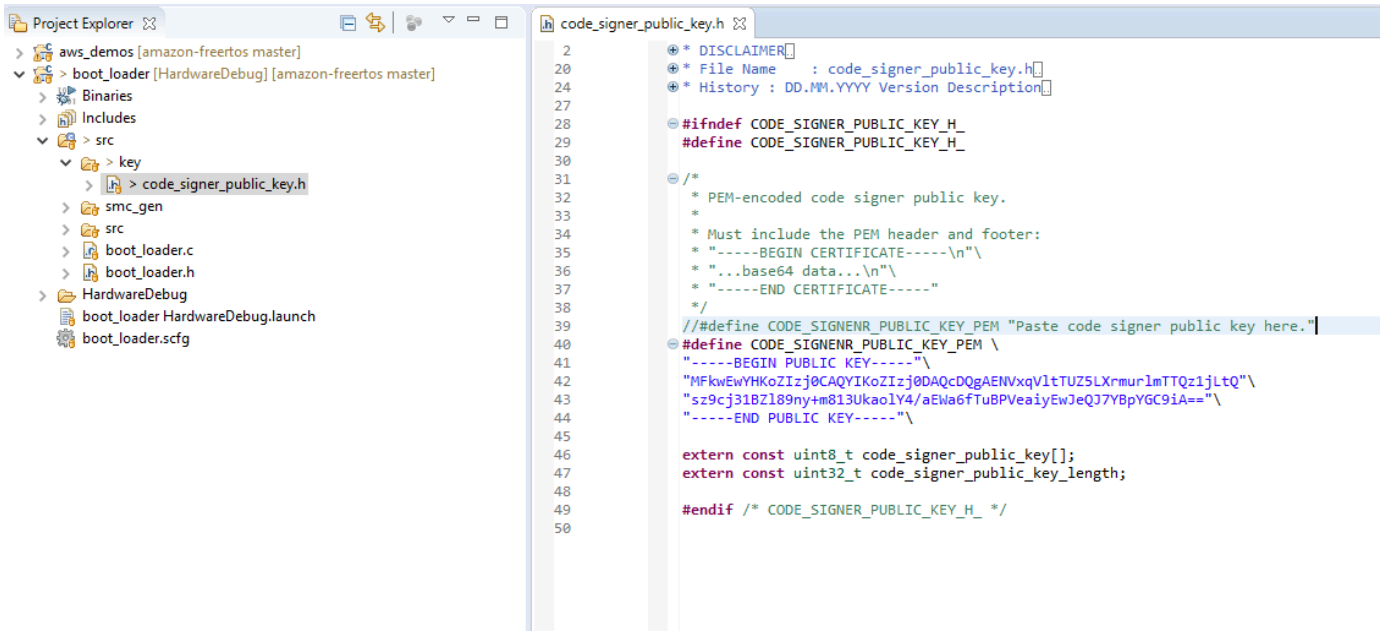
10. In the navigation pane, choose **Settings**. Choose the **Toolchain** tab, and then choose the toolchain **Version**.



Choose the **Tool Settings** tab, expand **Converter** and then choose **Output**. In the main window, make sure **Output hex file** is selected, and then choose the **Output file type**.



11. In the bootloader project, open `projects\renesas\rx65n-rsk\e2studio\boot_loader\src\key\code_signer_public_key.h` and input the public key. For information on how to create a public key, see How to implement FreeRTOS OTA by using Amazon Web Services on RX65N and section 7.3 "Generating ECDSA-SHA256 Key Pairs with OpenSSL" in Renesas MCU Firmware Update Design Policy.



Then build the project to create `boot_loader.mot`.

12. Open the `aws_demos` project.

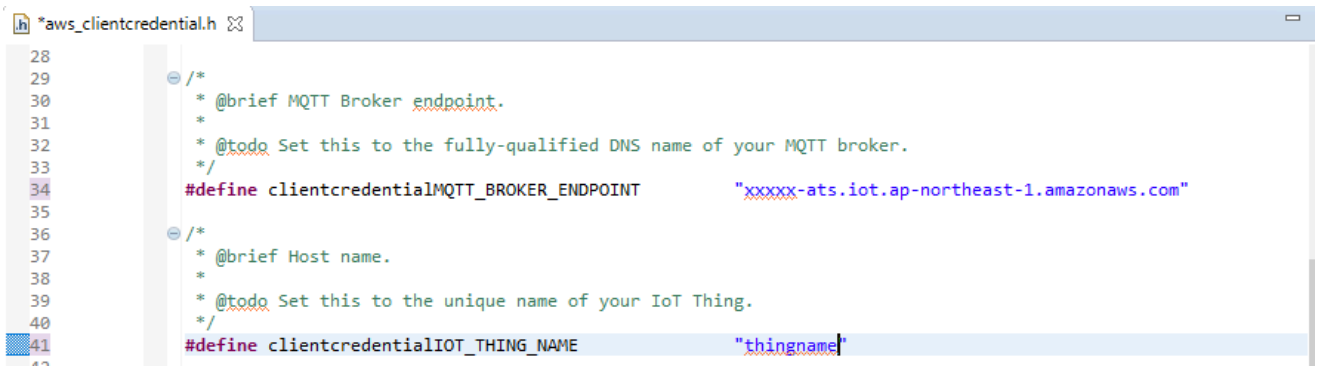
    a. Open the AWS IoT console.

    b. In the left navigation pane, choose **Settings**. Make a note of your custom endpoint in the **Device data endpoint** text box.

    c. Choose **Manage**, and then choose **Things**. Make a note of the AWS IoT thing name of your board.

    d. In the `aws_demos` project, open `demos/include/aws_clientcredential.h` and specify the following values.

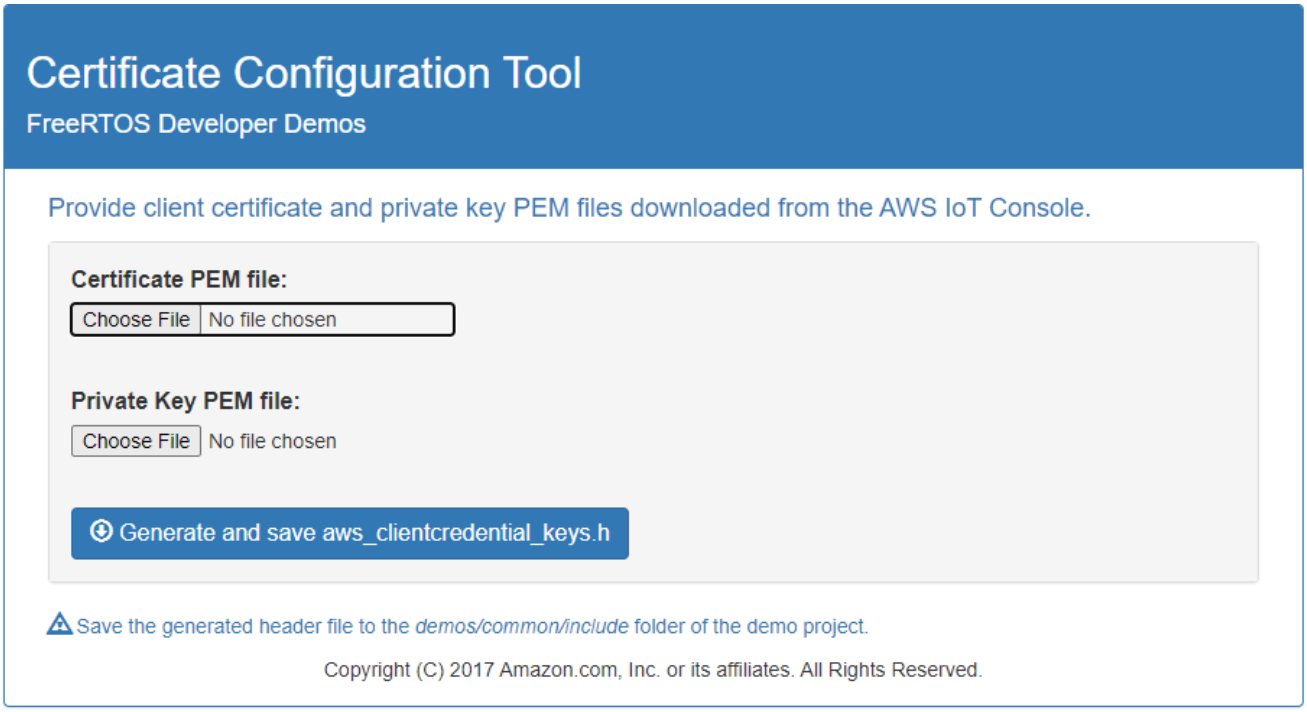
```
#define clientcredentialMQTT_BROKER_ENDPOINT[] = "Your AWS IoT endpoint";
#define clientcredentialIOT_THING_NAME "The AWS IoT thing name of your board"
```



e. Open the `tools/certificate_configuration/CertificateConfigurator.html` file.

f. Import the certificate PEM file and Private Key PEM file that you downloaded earlier.

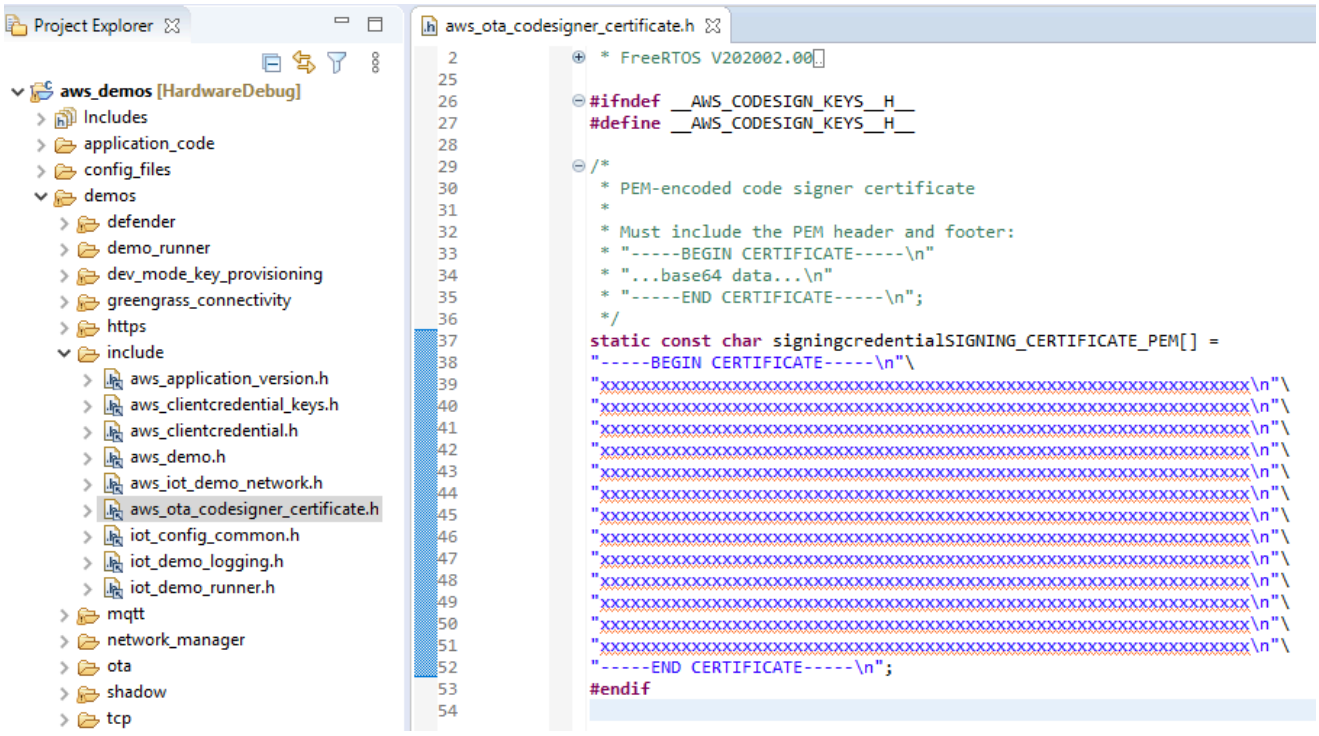
g. Choose **Generate and save aws_clientcredential_keys.h** and replace this file in the `demos/include/` directory.



h. Open the `vendors/renesas/boards/rx65n-rsk/aws_demos/config_files/ota_demo_config.h` file, and specify these values.

```
#define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
```

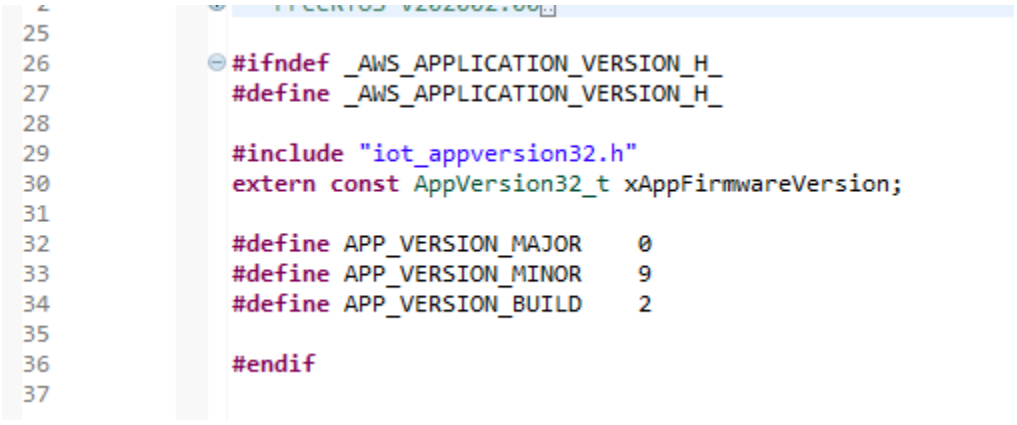
Where *your-certificate-key* is the value from the file `secp256r1.crt`. Remember to add "\" after each line in the certification. For more information on creating the `secp256r1.crt` file, see How to implement FreeRTOS OTA by using Amazon Web Services on RX65N and section 7.3 "Generating ECDSA-SHA256 Key Pairs with OpenSSL" in Renesas MCU Firmware Update Design Policy.



13. **Task A: Install the initial version of the firmware**

    a. Open the `vendors/renesas/boards/board/aws_demos/config_files/aws_demo_config.h` file, comment out `#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED`, and define either `CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED` or `CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED`.

    b. Open the `demos/include/ aws_application_version.h` file, and set the initial version of the firmware to `0.9.2`.



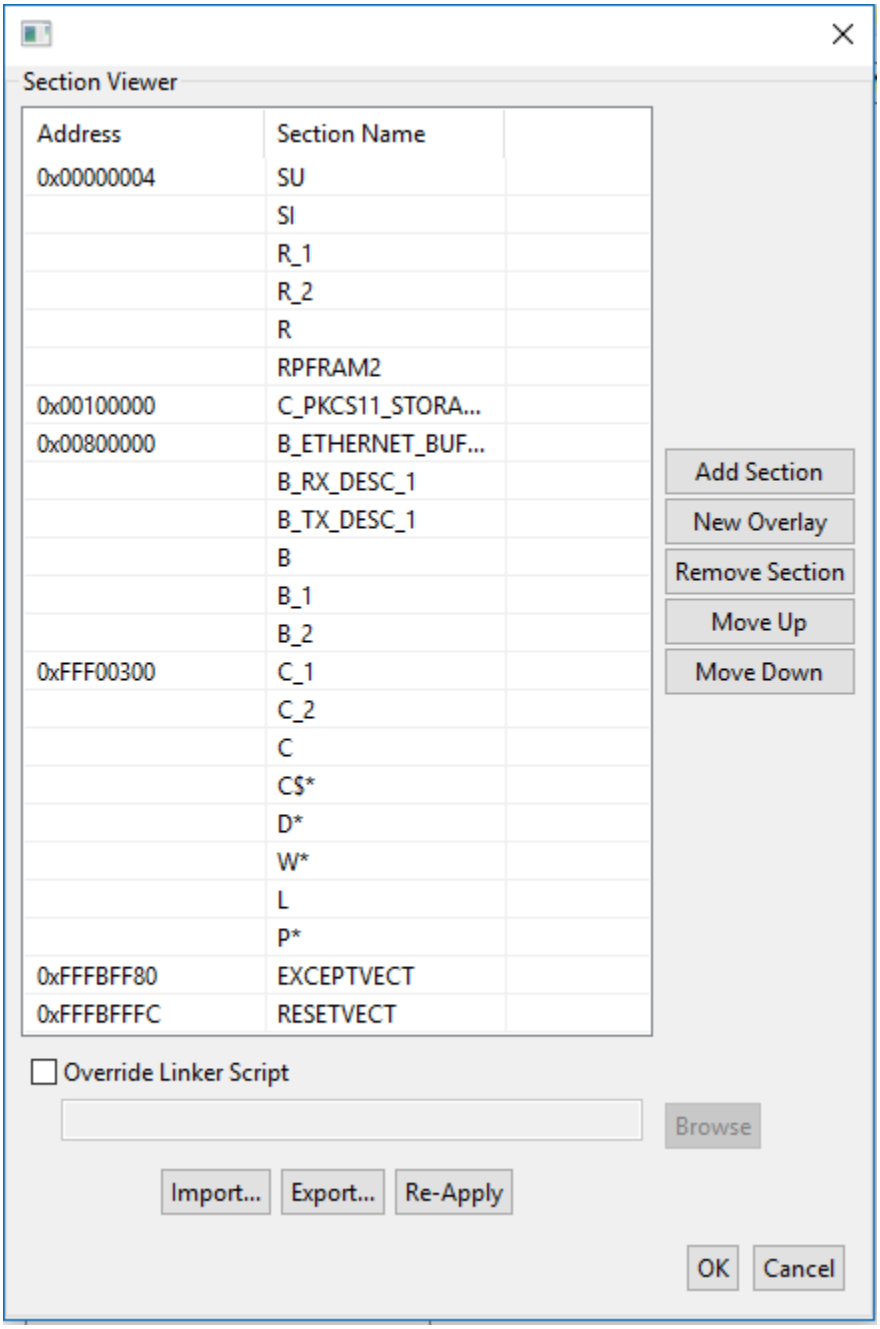
```
#ifndef _AWS_APPLICATION_VERSION_H_
#define _AWS_APPLICATION_VERSION_H_

#include "iot_appversion32.h"
extern const AppVersion32_t xAppFirmwareVersion;

#define APP_VERSION_MAJOR    0
#define APP_VERSION_MINOR    9
#define APP_VERSION_BUILD    2

#endif
```

    c. Change the following settings in the **Section Viewer**.



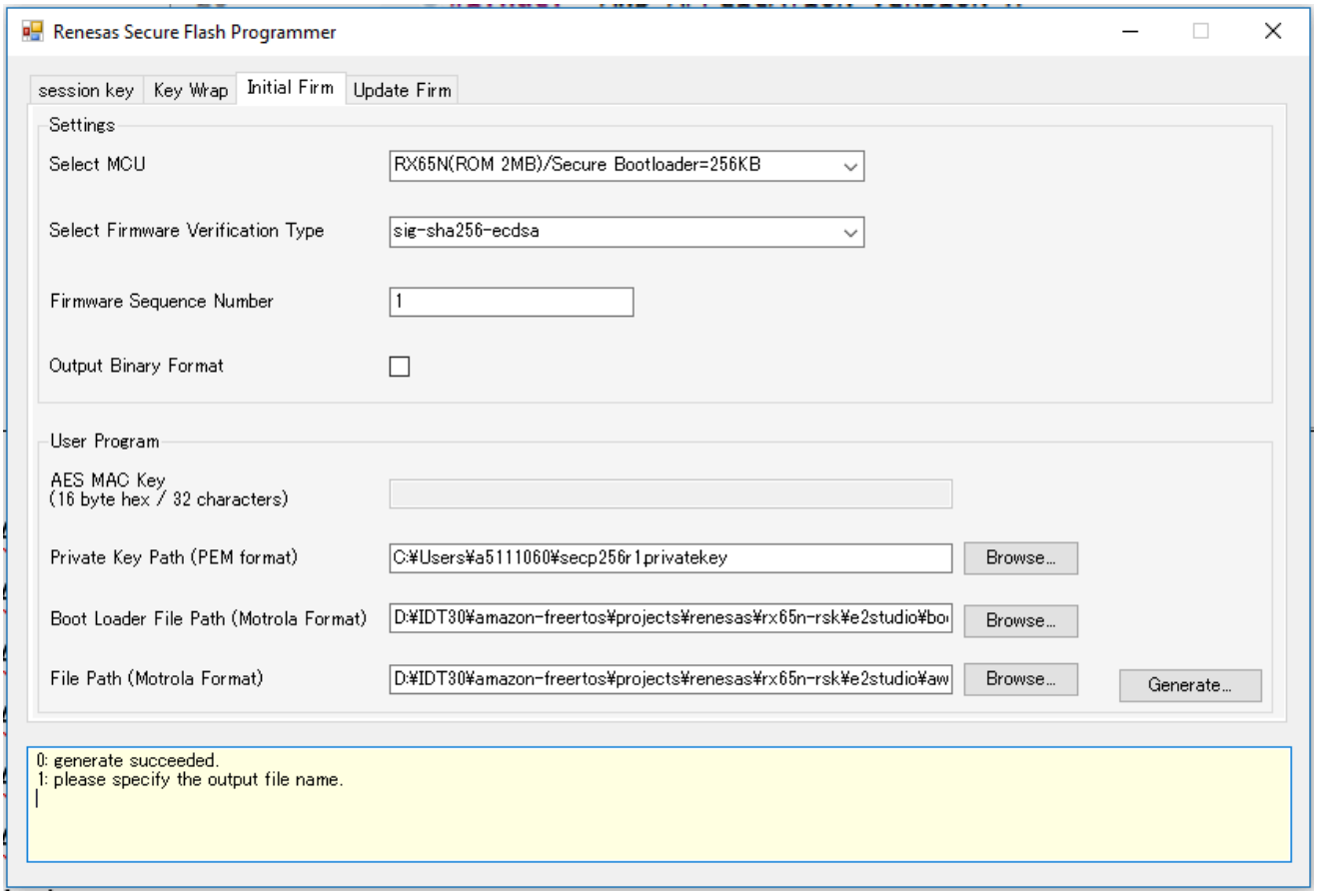
d. Choose **Build** to create the `aws_demos.mot` file.

14. Create the file `userprog.mot` with the Renesas Secure Flash Programmer. `userprog.mot` is a combination of `aws_demos.mot` and `boot_loader.mot`. You can flash this file to the RX65N-RSK to install the initial firmware.

    a. Download https://github.com/renesas/Amazon-FreeRTOS-Tools and open `Renesas Secure Flash Programmer.exe`.

    b. Choose the **Initial Firm** tab and then set the following parameters:

- **Private Key Path** – The location of `secp256r1.privatekey`.
- **Boot Loader File Path**– The location of `boot_loader.mot` (`projects\renesas\rx65n-rsk\e2studio\boot_loader\HardwareDebug`).
- **File Path** – The location of the `aws_demos.mot` (`projects\renesas\rx65n-rsk\e2studio\aws_demos\HardwareDebug`).



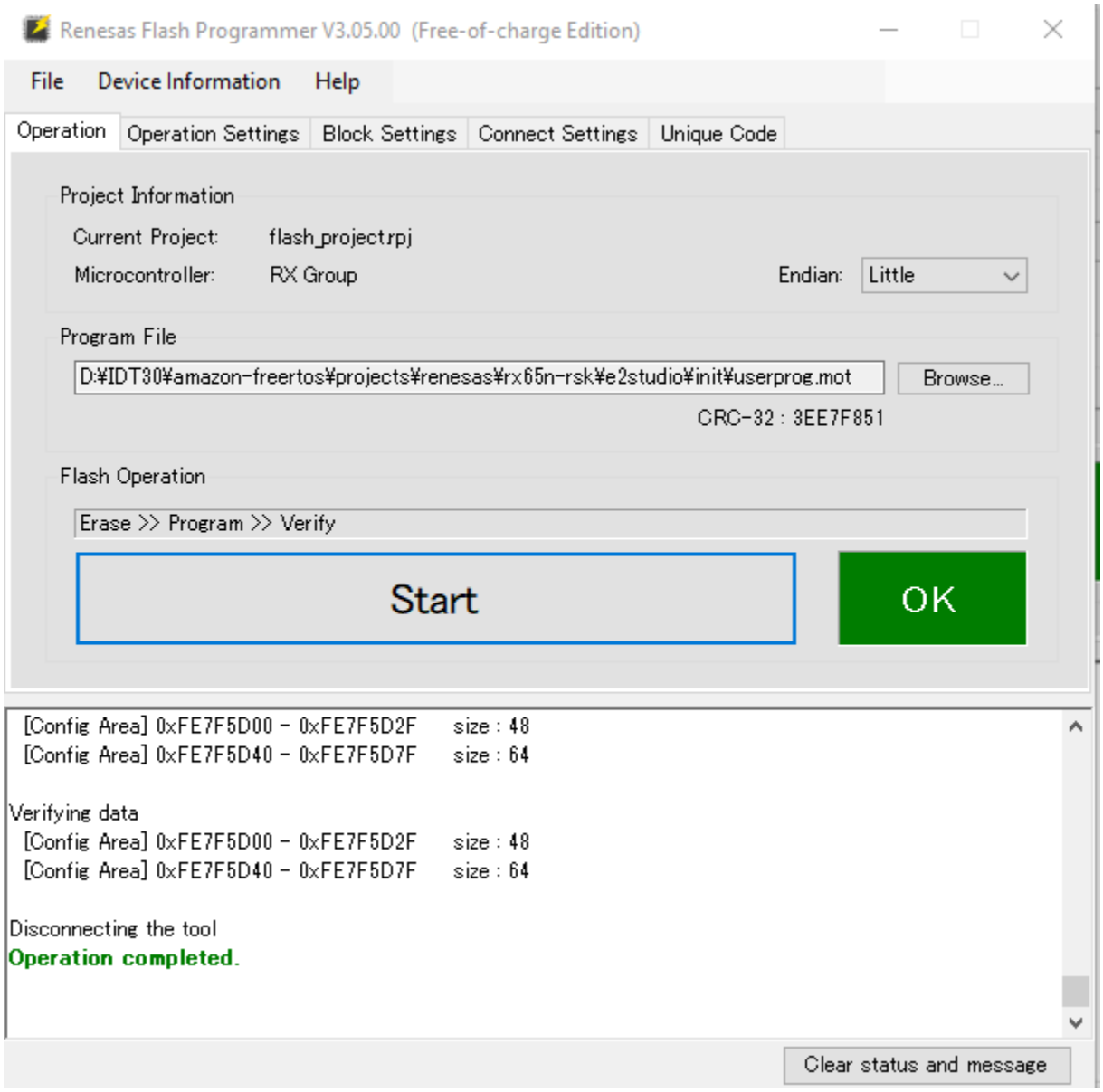
c. Create a directory named `init_firmware`, Generate `userprog.mot`, and save it to the `init_firmware` directory. Verify that the generate succeeded.

15. Flash the initial firmware on the RX65N-RSK.

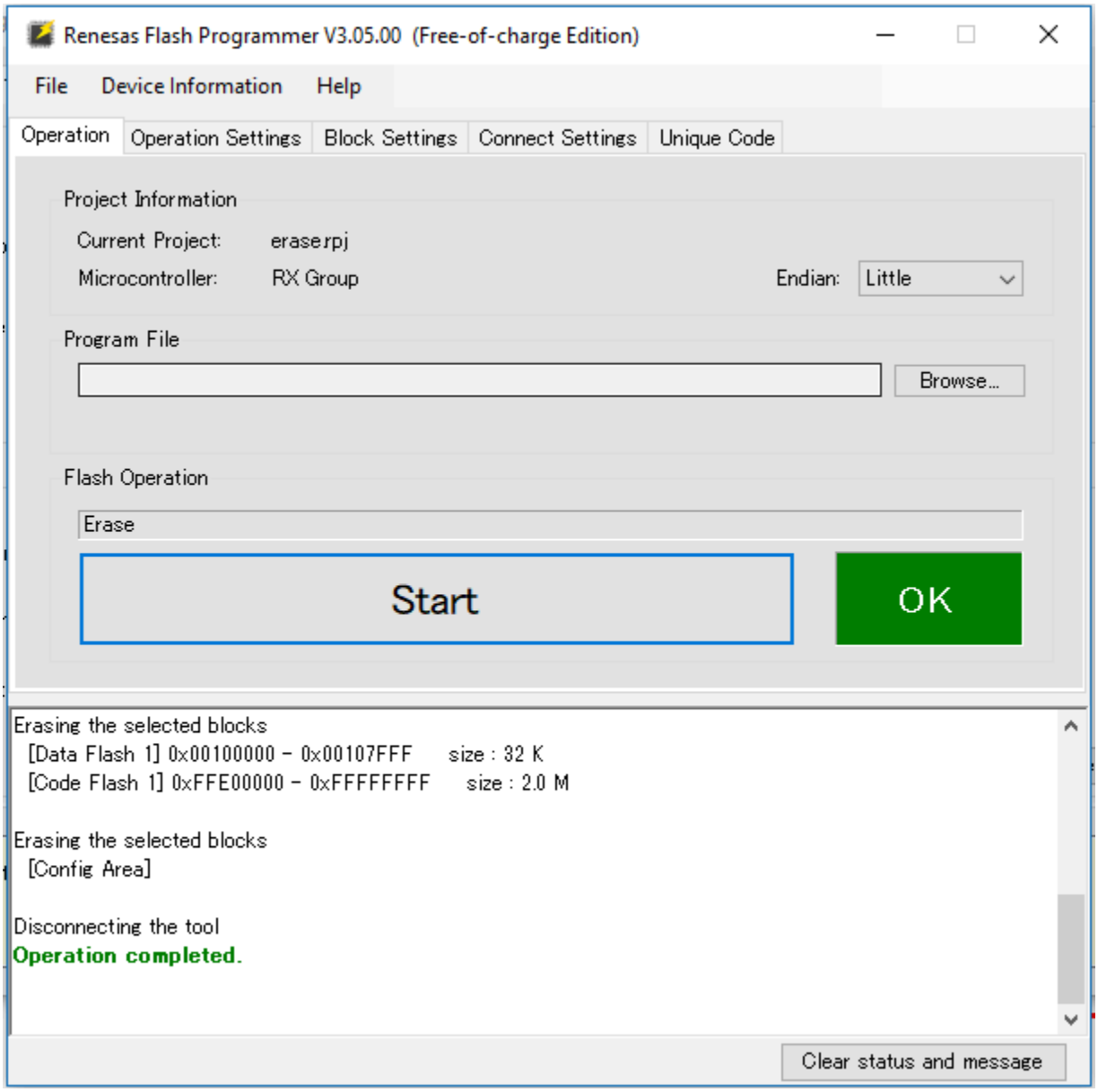
a. Download the latest version of the Renesas Flash Programmer (Programming GUI) from https://www.renesas.com/tw/en/products/software-tools/tools/programmer/renesas-flash-programmer-programming-gui.html.

b. Open the `vendors\renesas\rx_mcu_boards\boards\rx65n-rsk\aws_demos\flash_project\erase_from_bank\ erase.rpj` file to erase data on the bank.

c. Choose **Start** to erase the bank.



d. To flash `userprog.mot`, choose **Browse...** and navigate to the `init_firmware` directory, select the `userprog.mot` file and choose **Start**.



16. Version 0.9.2 (initial version) of the firmware was installed to your RX65N-RSK. The RX65N-RSK board is now listening for OTA updates. If you have opened Tera Term on your PC, you see something like the following when the initial firmware runs.

```
------------------------------------------------
RX65N secure boot program
------------------------------------------------
Checking flash ROM status.
bank 0 status = 0xff [LIFECYCLE_STATE_BLANK]
bank 1 status = 0xfc [LIFECYCLE_STATE_INSTALLING]
bank info = 1. (start bank = 0)
start installing user program.
```

```
copy secure boot (part1) from bank0 to bank1...OK
copy secure boot (part2) from bank0 to bank1...OK
update LIFECYCLE_STATE from [LIFECYCLE_STATE_INSTALLING] to [LIFECYCLE_STATE_VALID]
bank1(temporary area) block0 erase (to update LIFECYCLE_STATE)...OK
bank1(temporary area) block0 write (to update LIFECYCLE_STATE)...OK
swap bank...
-----------------------------------------------
RX65N secure boot program
-----------------------------------------------
Checking flash ROM status.
bank 0 status = 0xf8 [LIFECYCLE_STATE_VALID]
bank 1 status = 0xff [LIFECYCLE_STATE_BLANK]
bank info = 0. (start bank = 1)
integrity check scheme = sig-sha256-ecdsa
bank0(execute area) on code flash integrity check...OK
jump to user program
#0 1 [ETHER_RECEI] Deferred Interrupt Handler Task started
1 1 [ETHER_RECEI] Network buffers: 3 lowest 3
2 1 [ETHER_RECEI] Heap: current 234192 lowest 234192
3 1 [ETHER_RECEI] Queue space: lowest 8
4 1 [IP-task] InitializeNetwork returns OK
5 1 [IP-task] xNetworkInterfaceInitialise returns 0
6 101 [ETHER_RECEI] Heap: current 234592 lowest 233392
7 2102 [ETHER_RECEI] prvEMACHandlerTask: PHY LS now 1
8 3001 [IP-task] xNetworkInterfaceInitialise returns 1
9 3092 [ETHER_RECEI] Network buffers: 2 lowest 2
10 3092 [ETHER_RECEI] Queue space: lowest 7
11 3092 [ETHER_RECEI] Heap: current 233320 lowest 233320
12 3193 [ETHER_RECEI] Heap: current 233816 lowest 233120
13 3593 [IP-task] vDHCPProcess: offer c0a80a09ip
14 3597 [ETHER_RECEI] Heap: current 233200 lowest 233000
15 3597 [IP-task] vDHCPProcess: offer c0a80a09ip
16 3597 [IP-task] IP Address: 192.168.10.9
17 3597 [IP-task] Subnet Mask: 255.255.255.0
18 3597 [IP-task] Gateway Address: 192.168.10.1
19 3597 [IP-task] DNS Server Address: 192.168.10.1
20 3600 [Tmr Svc] The network is up and running
21 3622 [Tmr Svc] Write certificate...
22 3697 [ETHER_RECEI] Heap: current 232320 lowest 230904
23 4497 [ETHER_RECEI] Heap: current 226344 lowest 225944
24 5317 [iot_thread] [INFO ][DEMO][5317] ---------STARTING DEMO---------

25 5317 [iot_thread] [INFO ][INIT][5317] SDK successfully initialized.
```

```
26 5317 [iot_thread] [INFO ][DEMO][5317] Successfully initialized the demo. Network
 type for the demo: 4
27 5317 [iot_thread] [INFO ][MQTT][5317] MQTT library successfully initialized.
28 5317 [iot_thread] [INFO ][DEMO][5317] OTA demo version 0.9.2

29 5317 [iot_thread] [INFO ][DEMO][5317] Connecting to broker...

30 5317 [iot_thread] [INFO ][DEMO][5317] MQTT demo client identifier is rx65n-gr-
rose (length 13).
31 5325 [ETHER_RECEI] Heap: current 206944 lowest 206504
32 5325 [ETHER_RECEI] Heap: current 206440 lowest 206440
33 5325 [ETHER_RECEI] Heap: current 206240 lowest 206240
38 5334 [ETHER_RECEI] Heap: current 190288 lowest 190288
39 5334 [ETHER_RECEI] Heap: current 190088 lowest 190088
40 5361 [ETHER_RECEI] Heap: current 158512 lowest 158168
41 5363 [ETHER_RECEI] Heap: current 158032 lowest 158032
42 5364 [ETHER_RECEI] Network buffers: 1 lowest 1
43 5364 [ETHER_RECEI] Heap: current 156856 lowest 156856
44 5364 [ETHER_RECEI] Heap: current 156656 lowest 156656
46 5374 [ETHER_RECEI] Heap: current 153016 lowest 152040
47 5492 [ETHER_RECEI] Heap: current 141464 lowest 139016
48 5751 [ETHER_RECEI] Heap: current 140160 lowest 138680
49 5917 [ETHER_RECEI] Heap: current 138280 lowest 138168
59 7361 [iot_thread] [INFO ][MQTT][7361] Establishing new MQTT connection.
62 7428 [iot_thread] [INFO ][MQTT][7428] (MQTT connection 81cfc8, CONNECT operation
 81d0e8) Wait complete with result SUCCESS.
63 7428 [iot_thread] [INFO ][MQTT][7428] New MQTT connection 4e8c established.
64 7430 [iot_thread] [OTA_AgentInit_internal] OTA Task is Ready.
65 7430 [OTA Agent T] [prvOTAAgentTask] Called handler. Current State [Ready] Event
 [Start] New state [RequestingJob]
66 7431 [OTA Agent T] [INFO ][MQTT][7431] (MQTT connection 81cfc8) SUBSCRIBE
 operation scheduled.
67 7431 [OTA Agent T] [INFO ][MQTT][7431] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Waiting for operation completion.
68 7436 [ETHER_RECEI] Heap: current 128248 lowest 127992
69 7480 [OTA Agent T] [INFO ][MQTT][7480] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Wait complete with result SUCCESS.
70 7480 [OTA Agent T] [prvSubscribeToJobNotificationTopics] OK: $aws/things/rx65n-
gr-rose/jobs/$next/get/accepted
71 7481 [OTA Agent T] [INFO ][MQTT][7481] (MQTT connection 81cfc8) SUBSCRIBE
 operation scheduled.
72 7481 [OTA Agent T] [INFO ][MQTT][7481] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Waiting for operation completion.
```

```
73 7530 [OTA Agent T] [INFO ][MQTT][7530] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Wait complete with result SUCCESS.
74 7530 [OTA Agent T] [prvSubscribeToJobNotificationTopics] OK: $aws/things/rx65n-
gr-rose/jobs/notify-next
75 7530 [OTA Agent T] [prvRequestJob_Mqtt] Request #0
76 7532 [OTA Agent T] [INFO ][MQTT][7532] (MQTT connection 81cfc8) MQTT PUBLISH
 operation queued.
77 7532 [OTA Agent T] [INFO ][MQTT][7532] (MQTT connection 81cfc8, PUBLISH
 operation 818b80) Waiting for operation completion.
78 7552 [OTA Agent T] [INFO ][MQTT][7552] (MQTT connection 81cfc8, PUBLISH
 operation 818b80) Wait complete with result SUCCESS.
79 7552 [OTA Agent T] [prvOTAAgentTask] Called handler. Current State
 [RequestingJob] Event [RequestJobDocument] New state [WaitingForJob]
80 7552 [OTA Agent T] [prvParseJSONbyModel] Extracted parameter [ clientToken:
 0:rx65n-gr-rose ]
81 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: execution
82 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: jobId
83 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: jobDocument
84 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: afr_ota
85 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: protocols
86 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: files
87 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: filepath
99 7651 [ETHER_RECEI] Heap: current 129720 lowest 127304
100 8430 [iot_thread] [INFO ][DEMO][8430] State: Ready  Received: 1   Queued: 0
 Processed: 0   Dropped: 0
101 9430 [iot_thread] [INFO ][DEMO][9430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
102 10430 [iot_thread] [INFO ][DEMO][10430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
103 11430 [iot_thread] [INFO ][DEMO][11430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
104 12430 [iot_thread] [INFO ][DEMO][12430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
105 13430 [iot_thread] [INFO ][DEMO][13430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
106 14430 [iot_thread] [INFO ][DEMO][14430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
107 15430 [iot_thread] [INFO ][DEMO][15430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
```

17. **Task B: Update the version of your firmware**

    a. Open the `demos/include/aws_application_version.h`file and increment the `APP_VERSION_BUILD` token value to `0.9.3`.

b. Rebuild the project.

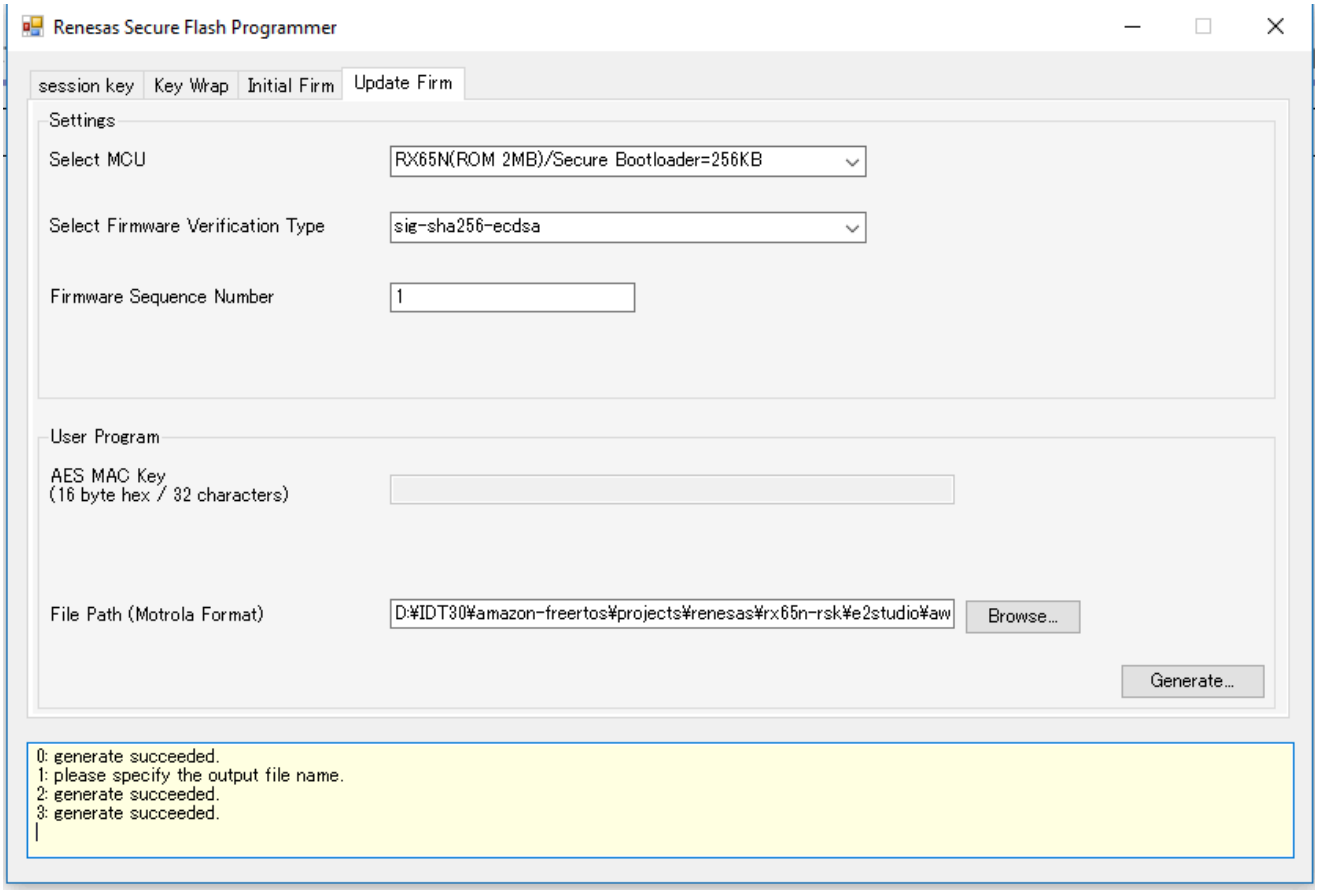
18. Create the `userprog.rsu` file with the Renesas Secure Flash Programmer to update the version of your firmware.

    a. Open the `Amazon-FreeRTOS-Tools\Renesas Secure Flash Programmer.exe` file.

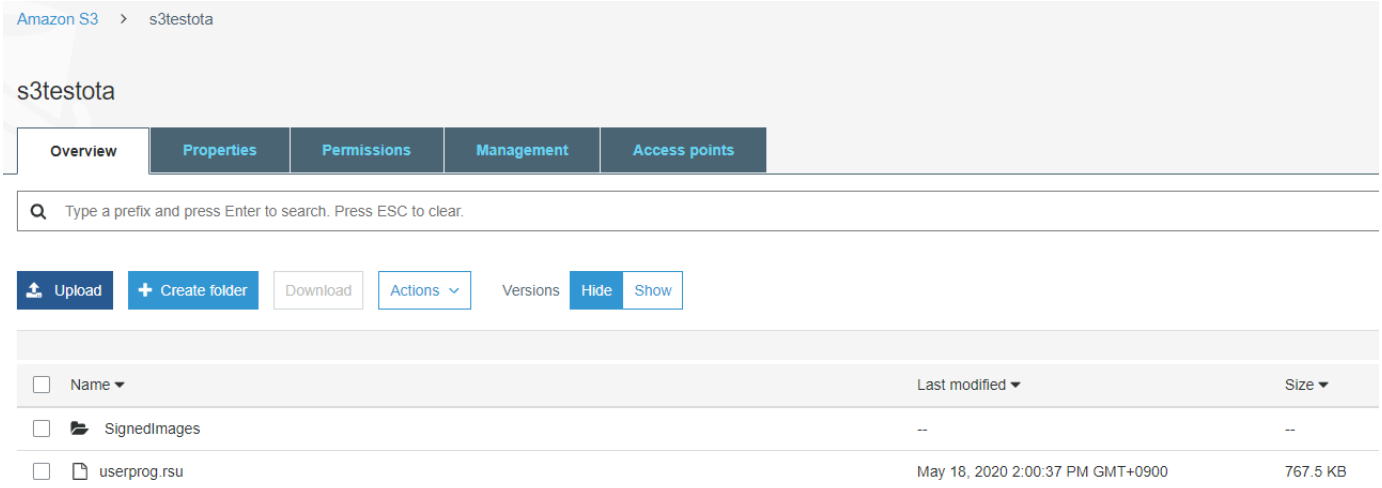
    b. Choose the **Update Firm** tab and set the following parameters:

    - **File Path** – The location of the `aws_demos.mot` file (`projects\renesas\rx65n-rsk\e2studio\aws_demos\HardwareDebug`).

    c. Create a directory named `update _firmware`. Generate `userprog.rsu` and save it to the `update_firmware` directory. Verify that the generate succeeded.



19. Upload the firmware update, `userproj.rsu`, into an Amazon S3 bucket as described in Create an Amazon S3 bucket to store your update.



20. Create a job to update firmware on the RX65N-RSK.

    AWS IoT Jobs is a service that notifies one or more connected devices of a pending Job. A job can be used to manage a fleet of devices, update firmware and security certificates on devices, or perform administrative tasks such as restarting devices and performing diagnostics.

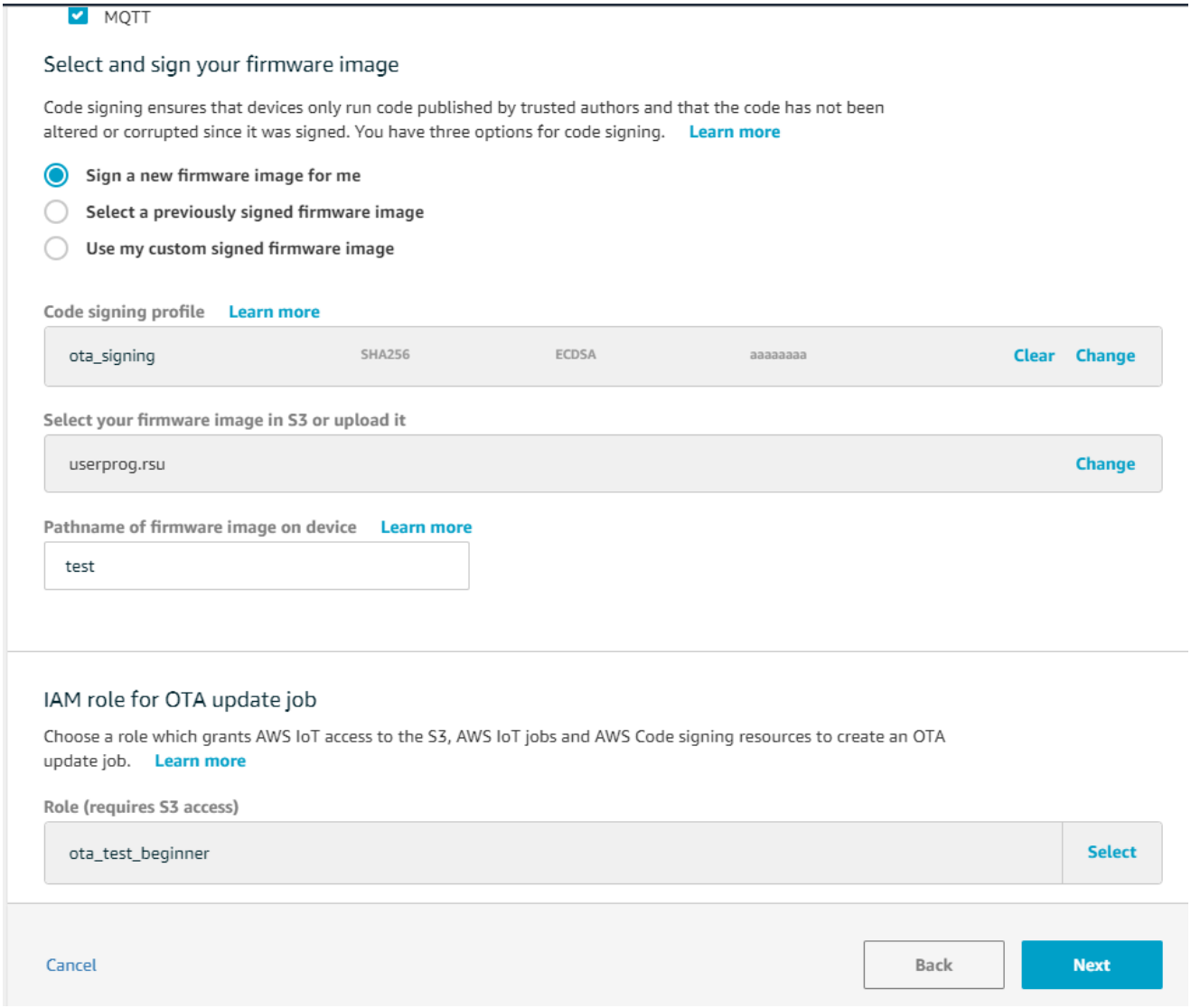
    a. Sign in to the AWS IoT console. In the navigation pane, choose **Manage**, and choose **Jobs**.

    b. Choose **Create a job**, then choose **Create OTA Update job**. Select a thing, then choose **Next**.

    c. Create a FreeRTOS OTA update job as follows:

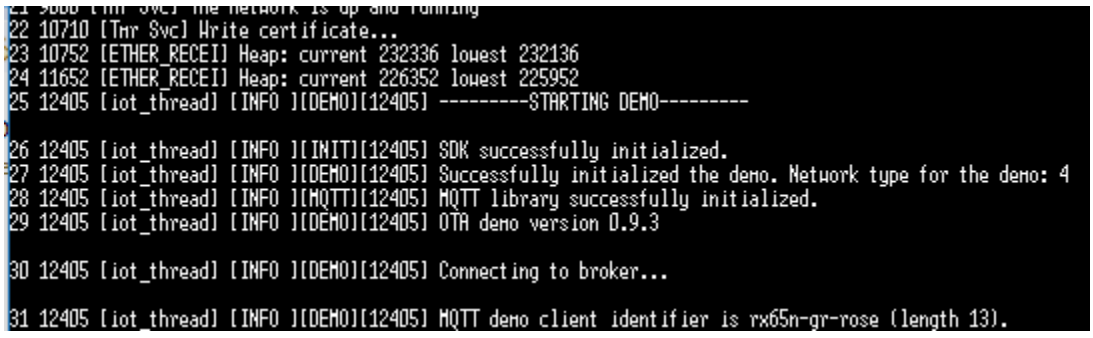
    - Choose **MQTT**.
    - Select the code signing profile you created in the previous section.
    - Select the firmware image that you uploaded to an Amazon S3 bucket.
    - For **Pathname of firmware image on device**, enter **`test`**.
    - Choose the IAM role that you created in the previous section.

    d. Choose **Next**.

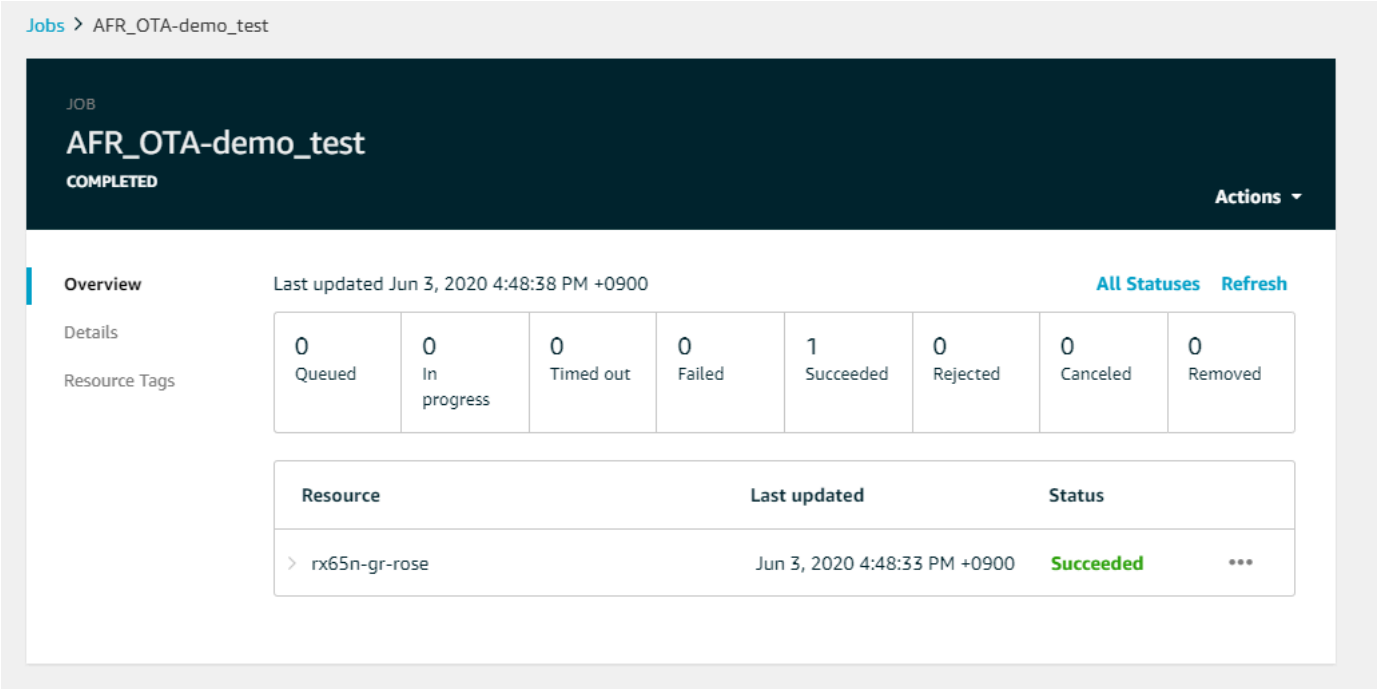


e. Enter an ID and then choose **Create**.

21. Reopen Tera Term to verify that the firmware was updated successfully to OTA demo version 0.9.3.



22. On the AWS IoT console, verify that the job status is **Succeeded**.



Tutorial: Perform OTA updates on Espressif ESP32 using FreeRTOS Bluetooth Low Energy

> **Important**
>
> This reference integration is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

This tutorial shows you how to update an Espressif ESP32 microcontroller that is connected to an MQTT Bluetooth Low Energy proxy on an Android device. It updates the device using AWS IoT Over-the-air (OTA) update jobs. The device connects to AWS IoT using Amazon Cognito credentials entered in the Android demo app. An authorized operator initiates the OTA update from the cloud. When the device connects through the Android demo app, the OTA update is initiated and the firmware is updated on the device.

FreeRTOS versions 2019.06.00 Major and later include Bluetooth Low Energy MQTT proxy support that can be used for Wi-Fi provisioning and secure connections to AWS IoT services. By using the Bluetooth Low Energy feature, you can build low-power devices that can be paired to a mobile device for connectivity without requiring Wi-Fi. Devices can communicate using MQTT by

connecting through Android or iOS Bluetooth Low Energy SDKs that use generic access profile (GAP) and generic attributes (GATT) profiles.

Here are the steps that we will follow to allow OTA updates over Bluetooth Low Energy:

1. **Configure storage:** Create an Amazon S3 bucket and policies and configure a user that can perform updates.
2. **Create a code-signing certificate:** Create a signing certificate and allow the user to sign firmware updates.
3. **Configure Amazon Cognito authentication:** Create a credential provider, user pool, and application access to the user pool.
4. **Configure FreeRTOS:** Set up Bluetooth Low Energy, client credentials, and the code-signing public certificate.
5. **Configure an Android app:** Set up credential provider, user pool, and deploy the application to an Android device.
6. **Run the OTA update script:** To initiate an OTA update, use the OTA update script.

For more information about how the updates work, see FreeRTOS Over-the-Air Updates. For additional information about how to set up the Bluetooth Low Energy MQTT proxy functionality, see the post Using Bluetooth Low Energy with FreeRTOS on Espressif ESP32 by Richard Kang.

Prerequisites

To perform the steps in this tutorial, you need the following resources:

- An ESP32 development board.
- A MicroUSB to USB A cable.
- An AWS account (the Free Tier is sufficient).
- An Android phone with Android v 6.0 or later and Bluetooth version 4.2 or later.

On your development computer you need:

- Sufficient disk space (~500 Mb) for the Xtensa toolchain and FreeRTOS source code and examples.
- Android Studio installed.
- The AWS CLI installed.

- Python3 installed.
- The boto3 AWS Software Developer Kit (SDK) for Python.

The steps in this tutorial assume that Xtensa toolchain, ESP-IDF, and FreeRTOS code are installed in the `/esp` directory in your home directory. You must add `~/esp/xtensa-esp32-elf/bin` to your `$PATH` variable.

Step 1: Configure storage

1. Create an Amazon S3 bucket to store your update with versioning enabled to hold the firmware images.
2. Create an OTA Update service role and add the following managed policies to the role:
   - AWSIotLogging
   - AWSIotRuleActions
   - AWSIotThingsRegistration
   - AWSFreeRTOSOTAUpdate
3. Create a user that can perform OTA updates. This user can sign and deploy firmware updates to IoT devices in the account, and has access to do OTA updates on all devices. Access should be limited to trusted entities.
4. Follow the steps to Create an OTA user policy and attach it to your user.

Step 2: Create the code-signing certificate

1. Create an Amazon S3 bucket with versioning enabled to hold the firmware images.
2. Create a code-signing certificate that can be used to sign the firmware. Note the certificate Amazon Resource Name (ARN) when the certificate is imported.

   ```
   aws acm import-certificate --profile=ota-update-user --certificate file://
   ecdsasigner.crt --private-key file://ecdsasigner.key
   ```

   Example output:

   ```
   {
   "CertificateArn": "arn:aws:acm:us-east-1:<account>:certificate/<certid>"
   }
   ```

You'll use the ARN later to create a signing profile. If you want, you can create the profile now with the following command:

```
aws signer put-signing-profile --profile=ota-update-user --profile-
name esp32Profile --signing-material certificateArn=arn:aws:acm:us-
east-1:account:certificate/certid --platform AmazonFreeRTOS-Default --signing-
parameters certname=/cert.pem
```

Example output:

```
{
"arn": "arn:aws:signer::<account>:/signing-profiles/esp32Profile"
}
```

Step 3: Amazon Cognito authentication configuration

Create an AWS IoT policy

1. Sign in to the AWS IoT console.
2. In the upper-right corner of the console, choose **My Account**. Under **Account Settings**, make a note of your 12-digit account ID.
3. In the left navigation pane, choose **Settings**. In **Device data endpoint**, make a note of the endpoint value. The endpoint should be something like `xxxxxxxxxxxxxx.iot.us-west-2.amazonaws.com`. In this example, the AWS Region is "us-west-2".
4. In the left navigation pane, choose **Secure**, choose **Policies**, and then choose **Create**. If you don't have any policies in your account, you will see the message "You don't have any policies yet" and you can choose **Create a policy**.
5. Enter a name for your policy, for example, "esp32_mqtt_proxy_iot_policy".
6. In the **Add statements** section, choose **Advanced mode**. Copy and paste the following JSON into the policy editor window. Replace `aws-account-id` with your account ID and `aws-region` with your Region (for example, "us-west-2").

   JSON

   ```
   {
       "Version": "2012-10-17",
   ```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Receive",
            "Resource": "arn:aws:iot:us-east-1:123456789012:*"
        }
    ]
}
```

7. Choose **Create**.

Create an AWS IoT thing

1. Sign in to the AWS IoT console.
2. In the left navigation pane, choose **Manage**, and then choose **Things**.
3. In the top-right corner, choose **Create**. If you don't have any things registered in your account, the message "You don’t have any things yet" is displayed and you can choose **Register a thing**.
4. On the **Creating AWS IoT things** page, choose **Create a single thing**.
5. On the **Add your device to the thing registry** page, enter a name for your thing (for example, "esp32-ble"). Only alphanumeric, hyphen (-), and underscore (_) characters are allowed. Choose **Next**.
6. On the **Add a certificate for your thing** page, under **Skip certificate and create thing**, choose **Create thing without certificate**. Because we use the BLE proxy mobile app that uses an Amazon Cognito credential for authentication and authorization, no device certificate is required.

Create an Amazon Cognito App Client

1. Sign in to the Amazon Cognito console.
2. In the top-right navigation banner, choose **Create a user pool**.
3. Enter the pool name (for example, "esp32_mqtt_proxy_user_pool").
4. Choose **Review defaults**.
5. In **App Clients**, choose **Add app client**, and then choose **Add an app client**.
6. Enter an app client name (for example "mqtt_app_client").
7. Make sure **Generate client secret** is selected.
8. Choose **Create app client**.
9. Choose **Return to pool details**.
10. On the **Review** page of the user pool, choose **Create pool**. You should see a message that says "Your user pool was created successfully." Make a note of the pool ID.
11. In the navigation pane, choose **App clients**.
12. Choose **Show Details**. Make a note of the app client ID and the app client secret.

Create an Amazon Cognito identity pool

1. Sign in to the Amazon Cognito console.
2. Choose **Create new identity pool**.
3. Enter a name for the identity pool (for example, "mqtt_proxy_identity_pool").
4. Expand **Authentication providers**.
5. Choose the **Cognito** tab.
6. Enter the user pool ID and app client ID that you noted in previous steps.
7. Choose **Create Pool**.
8. On the next page, to create new roles for authenticated and unauthenticated identities, choose **Allow**.
9. Make a note of the identity pool ID, which is in the format `us-east-1:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`.

Attach an IAM policy to the authenticated identity

1. Open the Amazon Cognito console.

2. Select the identity pool that you just created (for example, "mqtt_proxy_identity_pool").
3. Choose **Edit identity pool**.
4. Make a note of the IAM Role assigned to the authenticated role (for example, "Cognito_mqtt_proxy_identity_poolAuth_Role").
5. Open the IAM console.
6. In the navigation pane, choose **Roles**.
7. Search for the role assigned (for example, "Cognito_mqtt_proxy_identity_poolAuth_Role"), and then select it.
8. Choose **Add inline policy**, and then choose **JSON**.
9. Enter the following policy:

   JSON

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

10. Choose **Review Policy**.
11. Enter a policy name (for example, "mqttProxyCognitoPolicy").
12. Choose **Create policy**.

Step 4: Configure Amazon FreeRTOS

1. Download the latest version of the Amazon FreeRTOS code from the FreeRTOS GitHub repo.

2. To enable the OTA update demo, follow the steps in Getting started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT.
3. Make these additional modifications in the following files:

   a. Open `vendors/espressif/boards/esp32/aws_demos/config_files/aws_demo_config.h` and define `CONFIG_OTA_UPDATE_DEMO_ENABLED`.

   b. Open `vendors/espressif/boards/esp32/aws_demos/common/config_files/aws_demo_config.h` and change `democonfigNETWORK_TYPES` to `AWSIOT_NETWORK_TYPE_BLE`.

   c. Open `demos/include/aws_clientcredential.h` and enter your endpoint URL for `clientcredentialMQTT_BROKER_ENDPOINT`.

      Enter your thing name for `clientcredentialIOT_THING_NAME` (for example, "esp32-ble"). Certificates don't have to be added when you use Amazon Cognito credentials.

   d. Open `vendors/espressif/boards/esp32/aws_demos/config_files/aws_iot_network_config.h` and change `configSUPPORTED_NETWORKS` and `configENABLED_NETWORKS` to include only `AWSIOT_NETWORK_TYPE_BLE`.

   e. Open the `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/ota_demo_config.h` file, and enter your certificate.

      ```
      #define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
      ```

   The application should start up and print the demo version:

   ```
   11 13498 [iot_thread] [INFO ][DEMO][134980] Successfully initialized the demo.
    Network type for the demo: 2
   12 13498 [iot_thread] [INFO ][MQTT][134980] MQTT library successfully initialized.
   13 13498 [iot_thread] OTA demo version 0.9.20
   14 13498 [iot_thread] Creating MQTT Client...
   ```

Step 5: Configure an Android app

1. Download the Android Bluetooth Low Energy SDK and a sample app from the amazon-freertos-ble-android-sdk GitHub repo.

2. Open the file `app/src/main/res/raw/awsconfiguration.json` and fill in the Pool Id, Region, AppClientId, and AppClientSecret using the instructions in the following JSON sample.

```
{
  "UserAgent": "MobileHub/1.0",
  "Version": "1.0",
  "CredentialsProvider": {
    "CognitoIdentity": {
      "Default": {
        "PoolId": "Cognito->Manage Identity Pools->Federated Identities-
>mqtt_proxy_identity_pool->Edit Identity Pool->Identity Pool ID",
        "Region": "Your region (for example us-east-1)"
      }
    }
  },

  "IdentityManager": {
    "Default": {}
  },

  "CognitoUserPool": {
    "Default": {
      "PoolId": "Cognito-> Manage User Pools -> esp32_mqtt_proxy_user_pool ->
 General Settings -> PoolId",
      "AppClientId": "Cognito-> Manage User Pools -> esp32_mqtt_proxy_user_pool ->
 General Settings -> App clients ->Show Details",
      "AppClientSecret": "Cognito-> Manage User Pools -> esp32_mqtt_proxy_user_pool
 -> General Settings -> App clients ->Show Details",
      "Region": "Your region (for example us-east-1)"
    }
  }
}
```

3. Open `app/src/main/java/software/amazon/freertos/DemoConstants.java` and enter the policy name that you created earlier (for example, *`esp32_mqtt_proxy_iot_policy`*) and also the Region (for example, *`us-east-1`*).

4. Build and install the demo app.

   a. In Android Studio, choose **Build**, then **Make Module app**.

   b. Choose **Run**, then **Run app**. You can go to the logcat window pane in Android Studio to monitor log messages.

c. On the Android device, create an account from the login screen.

d. Create a user. If a user already exists, enter the credentials.

e. Allow the Amazon FreeRTOS Demo to access the device's location.

f. Scan for Bluetooth Low Energy devices.

g. Move the slider for the device found to **On**.

h. Press **y** on the serial port debug console for the ESP32.

i. Choose **Pair & Connect**.

5. The **More...** link becomes active after the connection is established. The connection state should change to "BLE_CONNECTED" in the Android device logcat when the connection is complete:

```
2019-06-06 20:11:32.160 23484-23497/software.amazon.freertos.demo I/FRD: BLE
 connection state changed: 0; new state: BLE_CONNECTED
```

6. Before the messages can be transmitted, the Amazon FreeRTOS device and the Android device negotiate the MTU. You should see the following output in logcat:

```
2019-06-06 20:11:46.720 23484-23497/software.amazon.freertos.demo I/FRD:
 onMTUChanged : 512 status: Success
```

7. The device connects to the app and starts sending MQTT messages using the MQTT proxy. To confirm that the device can communicate, make sure the MQTT_CONTROL characteristic data value changes to 01:

```
2019-06-06 20:12:28.752 23484-23496/software.amazon.freertos.demo D/FRD: <-<-<-
 Writing to characteristic: MQTT_CONTROL with data: 01
2019-06-06 20:12:28.839 23484-23496/software.amazon.freertos.demo D/FRD:
 onCharacteristicWrite for: MQTT_CONTROL; status: Success; value: 01
```

8. When the devices are paired, you will see a prompt on the ESP32 console. To enable BLE, press **y**. The demo won't function until you perform this step.

```
E (135538) BT_GATT: GATT_INSUF_AUTHENTICATION: MITM Required
W (135638) BT_L2CAP: l2cble_start_conn_update, the last connection update command
 still pending.
E (135908) BT_SMP: Value for numeric comparison = 391840
15 13588 [InputTask] Numeric comparison:391840
16 13589 [InputTask] Press 'y' to confirm
```

```
17 14078 [InputTask] Key accepted
W (146348) BT_SMP: FOR LE SC LTK IS USED INSTEAD OF STK
18 16298 [iot_thread] Connecting to broker...
19 16298 [iot_thread] [INFO ][MQTT][162980] Establishing new MQTT connection.
20 16298 [iot_thread] [INFO ][MQTT][162980] (MQTT connection 0x3ffd5754, CONNECT
 operation 0x3ffd586c) Waiting for operation completion.
21 16446 [iot_thread] [INFO ][MQTT][164450] (MQTT connection 0x3ffd5754, CONNECT
 operation 0x3ffd586c) Wait complete with result SUCCESS.
22 16446 [iot_thread] [INFO ][MQTT][164460] New MQTT connection 0x3ffc0ccc
 established.
23 16446 [iot_thread] Connected to broker.
```

Step 6: Run the OTA update script

1. To install the prerequisites, run the following commands:

   ```
   pip3 install boto3
   ```

   ```
   pip3 install pathlib
   ```

2. Increment the FreeRTOS application version in `demos/include/aws_application_version.h`.
3. Build a new .bin file.
4. Download the python script start_ota.py. To see the help contents for the script, run the following command in a terminal window:

   ```
   python3 start_ota.py -h
   ```

   You should see something like the following:

   ```
   usage: start_ota.py [-h] --profile PROFILE [--region REGION]
                       [--account ACCOUNT] [--devicetype DEVICETYPE] --name NAME
                       --role ROLE --s3bucket S3BUCKET --otasigningprofile
                       OTASIGNINGPROFILE --signingcertificateid
                       SIGNINGCERTIFICATEID [--codelocation CODELOCATION]
   Script to start OTA update
   optional arguments:
   -h, --help            show this help message and exit
   --profile PROFILE     Profile name created using aws configure
   ```

```
--region REGION       Region
--account ACCOUNT     Account ID
--devicetype DEVICETYPE thing|group
--name NAME           Name of thing/group
--role ROLE           Role for OTA updates
--s3bucket S3BUCKET   S3 bucket to store firmware updates
--otasigningprofile OTASIGNINGPROFILE
                      Signing profile to be created or used
--signingcertificateid SIGNINGCERTIFICATEID
                      certificate id (not arn) to be used
--codelocation CODELOCATION
                      base folder location (can be relative)
```

5. If you used the provided AWS CloudFormation template to create resources, run the following command:

```
python3 start_ota_stream.py --profile otausercf --name esp32-ble --role
 ota_ble_iot_role-sample --s3bucket afr-ble-ota-update-bucket-sample --
otasigningprofile abcd --signingcertificateid certificateid
```

   You should see the update start in the ESP32 debug console:

```
38 2462 [OTA Task] [prvParseJobDoc] Job was accepted. Attempting to start transfer.
---
49 2867 [OTA Task] [prvIngestDataBlock] Received file block 1, size 1024
50 2867 [OTA Task] [prvIngestDataBlock] Remaining: 1290
51 2894 [OTA Task] [prvIngestDataBlock] Received file block 2, size 1024
52 2894 [OTA Task] [prvIngestDataBlock] Remaining: 1289
53 2921 [OTA Task] [prvIngestDataBlock] Received file block 3, size 1024
54 2921 [OTA Task] [prvIngestDataBlock] Remaining: 1288
55 2952 [OTA Task] [prvIngestDataBlock] Received file block 4, size 1024
56 2953 [OTA Task] [prvIngestDataBlock] Remaining: 1287
57 2959 [iot_thread] State: Active  Received: 5   Queued: 5   Processed: 5
 Dropped: 0
```

6. When the OTA update is complete, the device restarts as required by the OTA update process. It then tries to connect using the updated firmware. If the upgrade has succeeded, the updated firmware is marked as active and you should see the updated version in the console:

```
13 13498 [iot_thread] OTA demo version 0.9.21
```

AWS IoT Device Shadow demo application

Important

This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

Introduction

This demo shows how to use the AWS IoT Device Shadow library to connect to the AWS Device Shadow service. It uses the coreMQTT library to establish an MQTT connection with TLS (Mutual Authentication) to the AWS IoT MQTT Broker and the coreJSON library parser to parse shadow documents received from the AWS Shadow service. The demo shows basic shadow operations, such as how to update a shadow document and how to delete a shadow document. The demo also shows how to register a callback function with the coreMQTT library to handle messages like the shadow `/update` and `/update/delta` messages that are sent from the AWS IoT Device Shadow service.

This demo is intended as a learning exercise only because the request to update the shadow document (state) and the update response are done by the same application. In a realistic production scenario, an external application would request an update of the state of the device remotely, even if the device is not currently connected. The device will acknowledge the update request when it is connected.

Note

To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS.

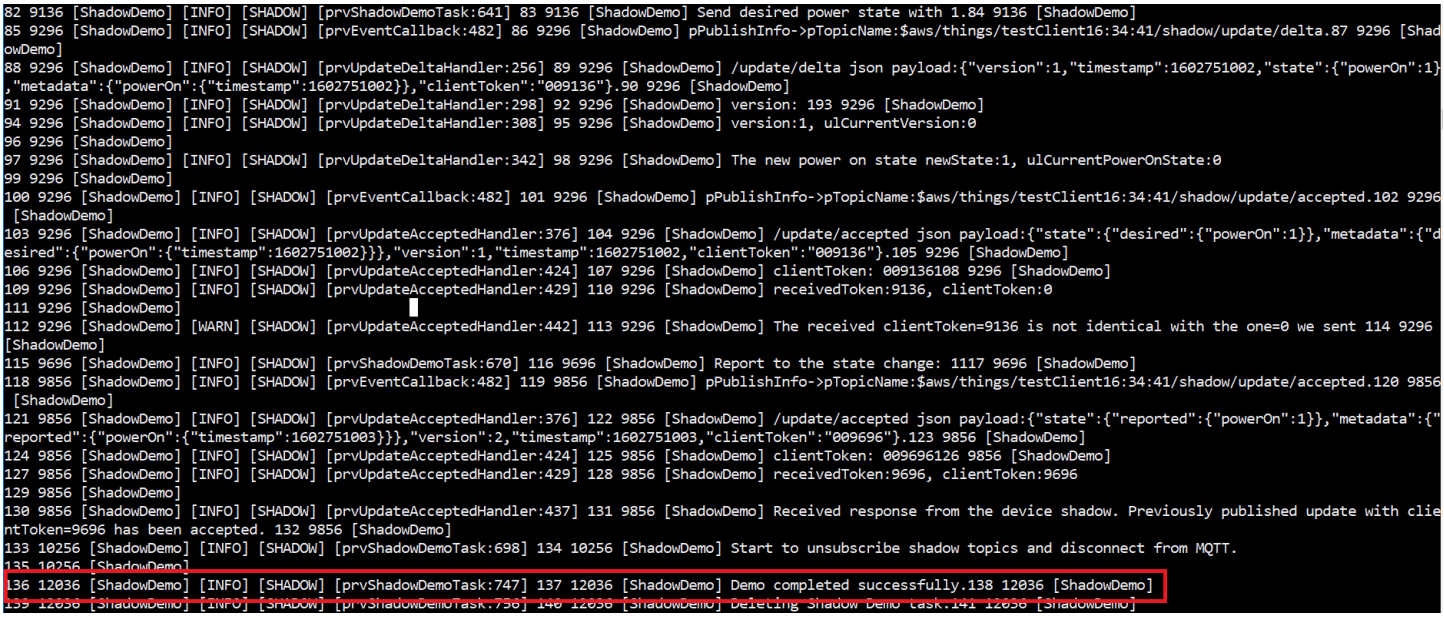
Functionality

The demo creates a single application task that loops through a set of examples that demonstrate shadow `/update` and `/update/delta` callbacks to simulate toggling a remote device's state. It sends a shadow update with the new `desired` state and waits for the device to change its `reported` state in response to the new `desired` state. In addition, a shadow `/update` callback is

used to print the changing shadow states. This demo also uses a secure MQTT connection to the AWS IoT MQTT Broker, and assumes there is a `powerOn` state in the device shadow.

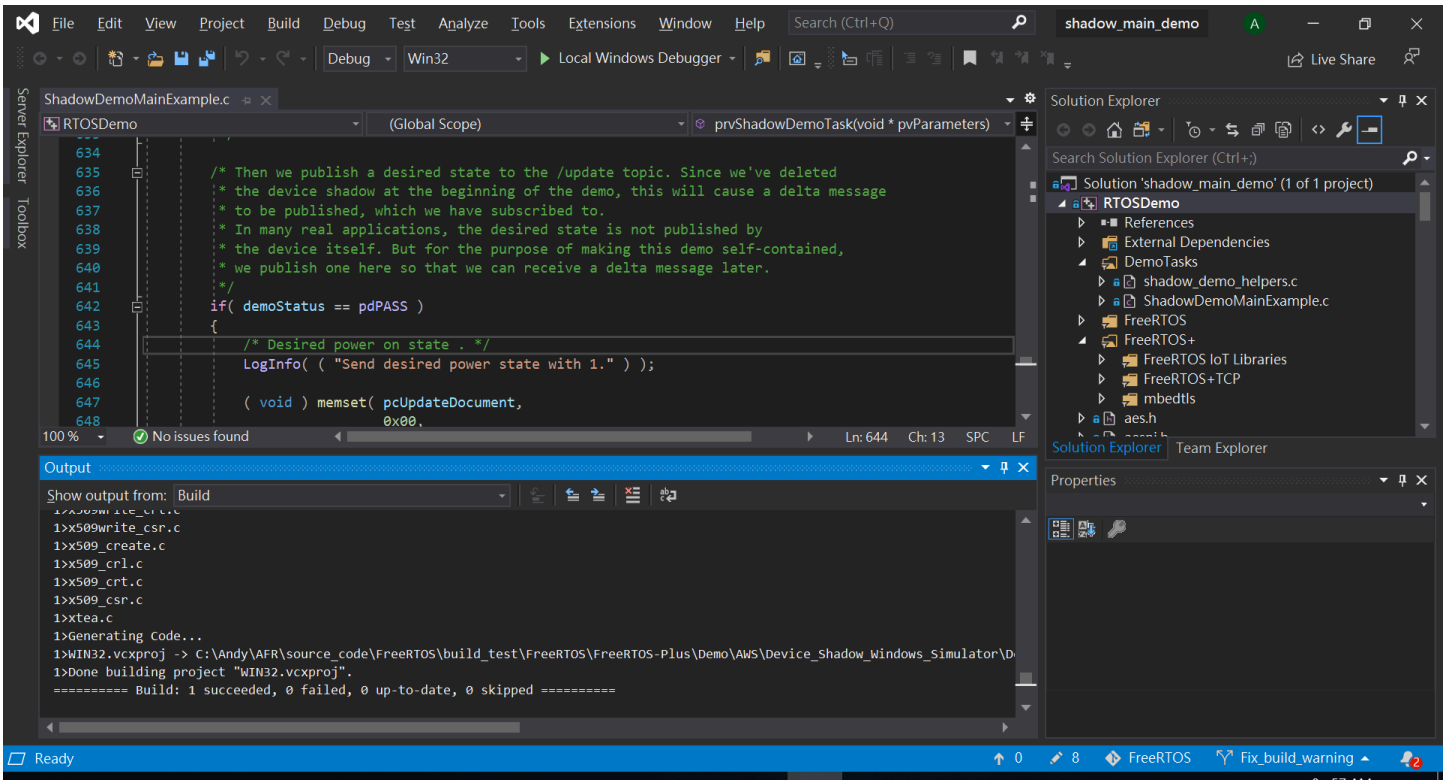
The demo performs the following operations:

1. Establish an MQTT connection by using the helper functions in `shadow_demo_helpers.c`.
2. Assemble MQTT topic strings for device shadow operations, using macros defined by the AWS IoT Device Shadow library.
3. Publish to the MQTT topic used for deleting a device shadow to delete any existing device shadow.
4. Subscribe to the MQTT topics for `/update/delta`, `/update/accepted` and `/update/rejected` using helper functions in `shadow_demo_helpers.c`.
5. Publish a desired state of `powerOn` using helper functions in `shadow_demo_helpers.c`. This will cause an `/update/delta` message to be sent to the device.
6. Handle incoming MQTT messages in `prvEventCallback`, and determine whether the message is related to the device shadow by using a function defined by the AWS IoT Device Shadow library (`Shadow_MatchTopic`). If the message is a device shadow `/update/delta` message, then the main demo function will publish a second message to update the reported state to `powerOn`. If an `/update/accepted` message is received, verify that it has the same `clientToken` as previously published in the update message. That will mark the end of the demo.



The demo can be found in the file *freertos*/demos/device_shadow_for_aws/shadow_demo_main.c or on GitHub.

The following screenshot shows the expected output when the demo succeeds.



Connect to the AWS IoT MQTT broker

To connect to the AWS IoT MQTT broker, we use the same method as `MQTT_Connect()` in the coreMQTT mutual authentication demo.

Delete the shadow document

To delete the shadow document, call `xPublishToTopic` with an empty message, using macros defined by the AWS IoT Device Shadow library. This uses `MQTT_Publish` to publish to the `/delete` topic. The following code section shows how this is done in the function `prvShadowDemoTask`.

```
/* First of all, try to delete any Shadow document in the cloud. */
returnStatus = PublishToTopic( SHADOW_TOPIC_STRING_DELETE( THING_NAME ),
                               SHADOW_TOPIC_LENGTH_DELETE( THING_NAME_LENGTH ),
                               pcUpdateDocument,
```

```
                                    0U );
```

Subscribe to shadow topics

Subscribe to the Device Shadow topics to receive notifications from the AWS IoT broker about shadow changes. The Device Shadow topics are assembled by macros defined in the Device Shadow library. The following code section shows how this is done in the `prvShadowDemoTask` function.

```
/* Then try to subscribe shadow topics. */
if( returnStatus == EXIT_SUCCESS )
{
    returnStatus = SubscribeToTopic(
                      SHADOW_TOPIC_STRING_UPDATE_DELTA( THING_NAME ),
                      SHADOW_TOPIC_LENGTH_UPDATE_DELTA( THING_NAME_LENGTH ) );
}

if( returnStatus == EXIT_SUCCESS )
{
    returnStatus = SubscribeToTopic(
                      SHADOW_TOPIC_STRING_UPDATE_ACCEPTED( THING_NAME ),
                      SHADOW_TOPIC_LENGTH_UPDATE_ACCEPTED( THING_NAME_LENGTH ) );
}

if( returnStatus == EXIT_SUCCESS )
{
    returnStatus = SubscribeToTopic(
                      SHADOW_TOPIC_STRING_UPDATE_REJECTED( THING_NAME ),
                      SHADOW_TOPIC_LENGTH_UPDATE_REJECTED( THING_NAME_LENGTH ) );
}
```

Send Shadow Updates

To send a shadow update, the demo calls `xPublishToTopic` with a message in JSON format, using macros defined by the Device Shadow library. This uses `MQTT_Publish` to publish to the `/delete` topic. The following code section shows how this is done in the `prvShadowDemoTask` function.

```
#define SHADOW_REPORTED_JSON    \
```

```
    "{"                          \
    "\"state\":{"                \
    "\"reported\":{"             \
    "\"powerOn\":%01d"           \
    "}"                          \
    "},"                         \
    "\"clientToken\":\"%06lu\"" \
    "}"
snprintf( pcUpdateDocument,
          SHADOW_REPORTED_JSON_LENGTH + 1,
          SHADOW_REPORTED_JSON,
           ( int ) ulCurrentPowerOnState,
           ( long unsigned ) ulClientToken );

xPublishToTopic( SHADOW_TOPIC_STRING_UPDATE( THING_NAME ),
                 SHADOW_TOPIC_LENGTH_UPDATE( THING_NAME_LENGTH ),
                 pcUpdateDocument,
                 ( SHADOW_DESIRED_JSON_LENGTH + 1 ) );
```

Handle shadow delta messages and shadow update messages

The user callback function, that was registered to the coreMQTT Client Library using the `MQTT_Init` function, will notify us about an incoming packet event. See the callback function prvEventCallback on GitHub.

The callback function confirms the incoming packet is of type `MQTT_PACKET_TYPE_PUBLISH`, and uses the Device Shadow Library API `Shadow_MatchTopic` to confirm that the incoming message is a shadow message.

If the incoming message is a shadow message with type `ShadowMessageTypeUpdateDelta`, then we call prvUpdateDeltaHandler to handle this message. The handler `prvUpdateDeltaHandler` uses the coreJSON library to parse the message to get the delta value for the `powerOn` state and compares this against the current device state maintained locally. If those are different, the local device state is updated to reflect the new value of the `powerOn` state from the shadow document.

If the incoming message is a shadow message with type `ShadowMessageTypeUpdateAccepted`, then we call prvUpdateAcceptedHandler to handle this message. The handler `prvUpdateAcceptedHandler` parses the message using the coreJSON library to get the `clientToken` from the message. This handler function checks that the client token from the JSON message matches the client token used by the application. If it doesn't match, the function logs a warning message.

Secure Sockets echo client demo

> **Important**
>
> This demo is hosted on the Amazon-FreeRTOS repository which is deprecated. We recommend that you start here when you create a new project. If you already have an existing FreeRTOS project based on the now deprecated Amazon-FreeRTOS repository, see the Amazon-FreeRTOS Github Repository Migration Guide.

The following example uses a single RTOS task. The source code for this example can be found at `demos/tcp/aws_tcp_echo_client_single_task.c`.

Before you begin, verify that you have downloaded FreeRTOS to your microcontroller and built and run the FreeRTOS demo projects. You can clone or download FreeRTOS from GitHub. See the README.md file for instructions.

To run the demo

> **Note**
>
> To set up and run the FreeRTOS demos, follow the steps in Get Started with FreeRTOS. The TCP server and client demos are currently not supported on the Cypress CYW943907AEVAL1F and CYW954907AEVAL1F Development Kits.

1. Follow the instructions in Setting Up the TLS Echo Server in the FreeRTOS Porting Guide.

   A TLS echo server should be running and listening on the port 9000.

   During the setup, you should have generated four files:

   - `client.pem` (client certificate)
   - `client.key` (client private key)
   - `server.pem` (server certificate)
   - `server.key` (server private key)
2. Use the tool `tools/certificate_configuration/CertificateConfigurator.html` to copy the client certificate (`client.pem`) and client private key (`client.key`) to `aws_clientcredential_keys.h`.

3. Open the `FreeRTOSConfig.h` file.
4. Set the `configECHO_SERVER_ADDR0`, `configECHO_SERVER_ADDR1`, `configECHO_SERVER_ADDR2`, and `configECHO_SERVER_ADDR3` variables to the four integers that make up the IP address where the TLS Echo Server is running.
5. Set the `configTCP_ECHO_CLIENT_PORT` variable to `9000`, the port where the TLS Echo Server is listening.
6. Set the `configTCP_ECHO_TASKS_SINGLE_TASK_TLS_ENABLED` variable to `1`.
7. Use the tool `tools/certificate_configuration/PEMfileToCString.html` to copy the server certificate (`server.pem`) to `cTlsECHO_SERVER_CERTIFICATE_PEM` in the file `aws_tcp_echo_client_single_task.c`.
8. Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED`, and define `CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED` or `CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED`.

The microcontroller and the TLS Echo Server should be on the same network. When the demo starts (`main.c`), you should see a log message that reads `Received correct string from echo server`.