FreeRTOS

User Guide



FreeRTOS: User Guide

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

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

Table of Contents

What Is FreeRTOS? ... 1
- FreeRTOS architecture ... 1
- FreeRTOS kernel ... 2
- FreeRTOS kernel fundamentals ... 3
  - FreeRTOS kernel scheduler ... 3
  - Memory management ... 3
  - Intertask coordination ... 4
  - Software timers ... 7
  - Low power support ... 7
  - `FreeRTOSConfig.h` ... 8
- FreeRTOS libraries ... 8
- Over-the-Air Updates ... 8
  - Tagging OTA resources ... 9
  - OTA update prerequisites ... 9
  - OTA tutorial ... 25
  - OTA Update Manager service ... 53
  - Integrating the OTA Agent into your application ... 53
  - OTA security ... 57
  - OTA troubleshooting ... 57
- Downloading FreeRTOS source code ... 65
- FreeRTOS console ... 65
- FreeRTOS console ... 65
  - Predefined FreeRTOS configurations ... 65
  - Custom FreeRTOS configurations ... 66
  - Quick connect workflow ... 67
  - Tagging configurations ... 67
- FreeRTOS-qualified hardware platforms ... 68
- Development workflow ... 69
- AWS IoT Device SDK for Embedded C ... 69
- Additional resources ... 70

Getting Started with FreeRTOS ... 71
- FreeRTOS demo application ... 71
- First steps ... 71
- Board-specific getting started guides ... 71
- Troubleshooting ... 71
- Developing FreeRTOS applications ... 71
- First steps ... 72
  - Setting up your AWS account and permissions ... 72
  - Registering your MCU board with AWS IoT ... 73
  - Downloading FreeRTOS ... 75
  - Configuring the FreeRTOS demos ... 75
- Developer-mode key provisioning ... 77
  - Introduction ... 77
  - Option #1: private key import from AWS IoT ... 77
  - Option #2: onboard private key generation ... 77
- Troubleshooting ... 79
  - General getting started troubleshooting tips ... 79
  - Installing a terminal emulator ... 79
- Using CMake with FreeRTOS ... 80
  - Prerequisites ... 81
  - Developing FreeRTOS applications with third-party code editors and debugging tools ... 81
  - Building FreeRTOS with CMake ... 82
- Board-specific getting started guides ... 86
  - Cypress CYW943907AEVAL1F Development Kit ... 87

Cypress CYW954907AEVAL1F Development Kit 90
Microchip ATECC608A Secure Element with Windows simulator 93
Espressif ESP32-DevKitC and the ESP-WROVER-KIT 97
Espressif ESP32-WROOM-32SE (Preview) 112
Infineon XMC4800 IoT Connectivity Kit 117
Infineon OPTIGA Trust X and XMC4800 IoT Connectivity Kit 121
Marvell MW320 AWS IoT Starter Kit 126
Marvell MW322 AWS IoT Starter Kit 136
MediaTek MT7697Hx development kit 147
Microchip Curiosity PIC32MZ EF 151
Nordic nRF52840-DK 154
Nuvoton NuMaker-IoT-M487 158
NXP LPC54018 IoT Module 164
Renesas Starter Kit+ for RX65N-2MB 167
STMicroelectronics STM32L4 Discovery Kit IoT Node 170
Texas Instruments CC3220SF-LAUNCHXL 172
Windows Device Simulator 176
Xilinx Avnet MicroZed Industrial IoT Kit 178
FreeRTOS Libraries 185
FreeRTOS porting libraries 185
FreeRTOS application libraries 190
FreeRTOS common libraries 193
Configuring the FreeRTOS libraries 194
Common libraries 195
Atomic operations 195
Linear Containers 195
Logging 196
Static Memory 196
Task Pool 196
Bluetooth Low Energy 199
Overview 199
Architecture 199
Dependencies and requirements 201
Library configuration file 201
Optimization 202
Usage restrictions 202
Initialization 203
API reference 204
Example usage 204
Porting 206
Mobile SDKs for FreeRTOS Bluetooth devices 208
AWS IoT Device Defender 210
Overview 210
Dependencies and requirements 210
Troubleshooting 211
Developer support 211
Usage restrictions 211
Initialization 212
FreeRTOS Device Defender API 212
Example usage 212
AWS IoT Device Shadow 212
Overview 212
Dependencies and requirements 212
API reference 213
Example usage 213
AWS IoT Greengrass 214
Overview 214

Dependencies and requirements ... 214
API reference ... 215
Example usage ... 215
MQTT (v2.0.0) ... 216
Overview ... 216
Dependencies and requirements ... 216
Features ... 217
API reference ... 217
Example usage ... 217
MQTT (v1.0.0) ... 218
Overview ... 218
Dependencies and requirements ... 218
Features ... 218
Major configurations ... 219
Optimization ... 219
Developer support ... 220
Initialization ... 221
API reference ... 221
Porting ... 221
HTTPS ... 221
Overview ... 221
Dependencies and requirements ... 221
Features ... 222
API reference ... 222
OTA Agent ... 222
Overview ... 222
Features ... 222
API reference ... 223
Example usage ... 223
Porting ... 223
Public Key Cryptography Standard (PKCS) #11 ... 224
Overview ... 224
Features ... 224
Asymmetric cryptosystem support ... 225
Porting ... 226
Secure Sockets ... 226
Overview ... 226
Dependencies and requirements ... 227
Features ... 227
Troubleshooting ... 227
Developer support ... 228
Usage restrictions ... 228
Initialization ... 228
API reference ... 228
Example usage ... 228
Porting ... 230
Transport Layer Security ... 230
Wi-Fi ... 230
Overview ... 230
Dependencies and requirements ... 230
Features ... 231
Configuration ... 232
Initialization ... 232
API reference ... 233
Example usage ... 233
Porting ... 234
Common I/O ... 234

FreeRTOS Demos 236
Running the FreeRTOS demos 236
Configuring the demos 236
Bluetooth Low Energy 236
Overview 236
Prerequisites 237
Common components 239
MQTT over Bluetooth Low Energy 243
Wi-Fi provisioning 245
Generic Attributes Server 247
Bootloader for the Microchip Curiosity PIC32MZEF 248
Bootloader states 248
Flash device 249
Application image structure 250
Image header 250
Image descriptor 251
Image trailer 252
Bootloader configuration 252
Building the bootloader 253
AWS IoT Device Defender 253
AWS IoT Greengrass 254
Amazon EC2 256
Over-the-air updates 257
Texas Instruments CC3220SF-LAUNCHXL 260
Microchip Curiosity PIC32MZEF 262
Espressif ESP32 266
HTTPS 266
Overview 266
Usage instructions 267
AWS IoT Device Shadow 267
Secure Sockets 269
Using AWS IoT Device Tester for FreeRTOS 270
Supported versions of IDT for FreeRTOS 271
Latest version of IDT for FreeRTOS 271
Earlier IDT versions 272
Unsupported IDT versions 273
Prerequisites 274
Download FreeRTOS 274
Download IDT for FreeRTOS 274
Create and configure an AWS account 274
AWS IoT Device Tester managed policy 276
(Optional) Install the AWS Command Line Interface 276
Preparing to test your microcontroller board for the first time 277
Add library porting layers 277
Configure your AWS credentials 277
Create a device pool in IDT for FreeRTOS 277
Configure build, flash, and test settings 280
Running Bluetooth Low Energy tests 286
Prerequisites 286
Raspberry Pi setup 286
FreeRTOS device setup 288
Running the BLE tests 288
Troubleshooting BLE tests 289
Running the FreeRTOS qualification suite 289
IDT for FreeRTOS commands 291
Test for re-qualification 292
Test suite versions 292

Understanding results and logs 293
Viewing results 293
Troubleshooting 295
Troubleshooting device configuration 295
Troubleshooting timeout errors 301
Support policy 301
Security in AWS 302
Identity and Access Management 302
Audience 303
Authenticating with identities 303
Managing access using policies 305
Learn more 306
How AWS services work with IAM 306
Identity-based policy examples 309
Troubleshooting 311
Compliance validation 313
Resilience 313
Infrastructure security 313

What Is FreeRTOS?

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

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

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

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

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

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

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

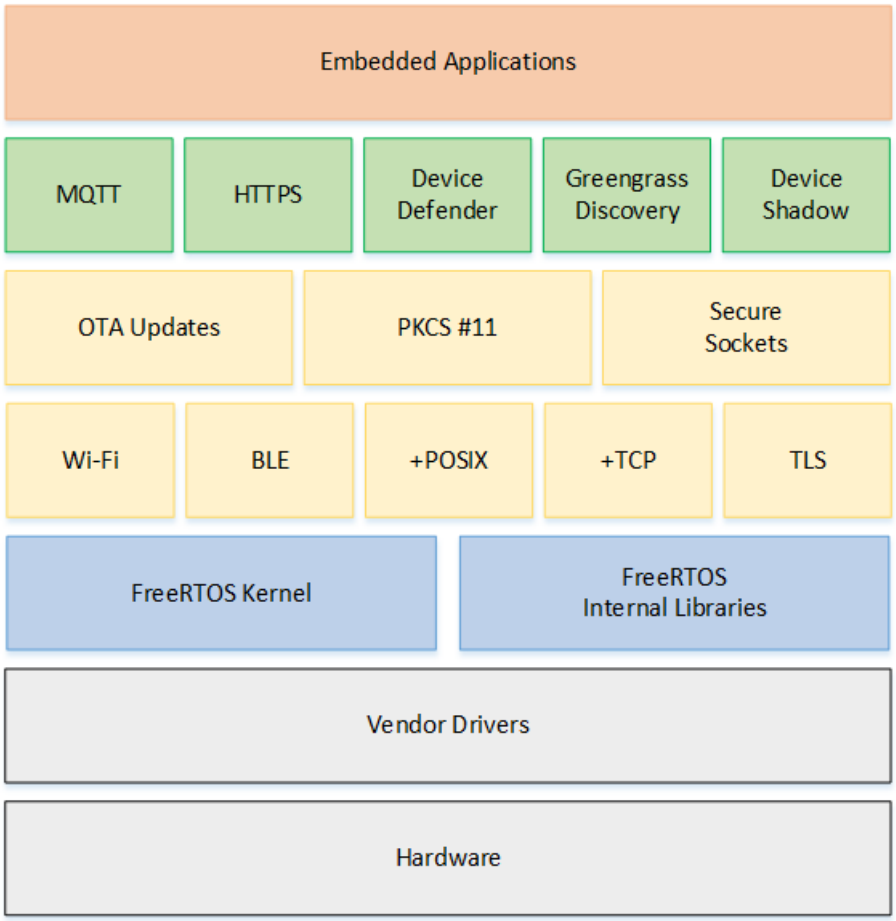
Previously, FreeRTOS releases used semantic versioning for major releases. Although it has moved to date-based versioning (FreeRTOS 1.4.8 updated to FreeRTOS 201906.00), the FreeRTOS kernel and each individual FreeRTOS library still retain semantic versioning. In semantic versioning, the version number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

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

FreeRTOS architecture

FreeRTOS is typically flashed to devices as a single compiled image with all of the components required for device applications. This image combines functionality for the applications written by the embedded

developer, the software libraries provided by Amazon, the FreeRTOS kernel, and drivers and board support packages (BSPs) for the hardware platform. Independent of the individual microcontroller being used, embedded application developers can expect the same standardized interfaces to the FreeRTOS kernel and all FreeRTOS software libraries.



FreeRTOS kernel

The FreeRTOS kernel is a real-time operating system that supports numerous architectures and is ideal for building embedded microcontroller applications. The kernel provides:

- A multitasking scheduler.
- Multiple memory allocation options (including the ability to create statically allocated systems).
- Inter-task coordination primitives, including task notifications, message queues, multiple types of semaphores, and stream and message buffers.

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 Quick Start Guide and the more in-depth Mastering the FreeRTOS Real Time Kernel. For more information about the FreeRTOS kernel in this guide, see FreeRTOS kernel fundamentals (p. 3).

FreeRTOS kernel fundamentals

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

- A multitasking scheduler.
- Multiple memory allocation options (including the ability to create completely statically-allocated systems).
- Intertask coordination primitives, including task notifications, message queues, multiple types of semaphore, and stream and message buffers.

The FreeRTOS kernel never performs non-deterministic operations, such as walking a linked list, inside a critical section or interrupt. The FreeRTOS kernel includes an efficient software timer implementation that does not use any CPU time unless a timer 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 intertask and interrupt-to-task signaling scenarios.

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

FreeRTOS kernel scheduler

An embedded application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context, with no dependency on other tasks. Only one task 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.

Memory management

This section provides information about kernel memory allocation and application memory management.

Kernel memory allocation

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

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

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

- They might not be available on embedded systems.
- They take up valuable code space.

- They are not typically thread-safe.
- They are not deterministic.

For these reasons, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, 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()`.

Application memory management

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

The FreeRTOS kernel includes five heap implementations:

`heap_1`

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.

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` (p. 8). 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` (p. 8).

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` (p. 8). 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` (p. 8).

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.

Kernel configuration

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.

FreeRTOS libraries

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 (p. 8).

For more information, see FreeRTOS Libraries.

FreeRTOS Over-the-Air 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 (p. 53) 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 OTA Agent library (p. 222) 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 (p. 22) or the Prerequisites for OTA updates using MQTT (p. 20).
- Create an Amazon S3 bucket to store your update (p. 9).
- Create an OTA Update service role (p. 10).
- Create an OTA user policy (p. 11).
- Create a code-signing certificate (p. 13).
- If you are using Code Signing for AWS IoT, Grant access to code signing for AWS IoT (p. 19).
- Download FreeRTOS with the OTA library (p. 20).

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. Type a bucket name, and then choose **Next**.

4. Select **Versioning** to keep all versions in the same bucket, and then choose **Next**.
5. Choose **Next** to accept the default permissions.
6. Choose **Create bucket**.

For more information about Amazon S3, see Amazon Simple Storage Service Console 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: Tags**.
8. Choose **Next: Review**.
9. Enter a role name and description, and then choose **Create role**.

For more information about IAM roles, see IAM Roles.

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:

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::your_account_id: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.

```
{
    "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 IAM user permission to perform over-the-air updates. Your IAM 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 Streaming service.
- 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 (p. 19).
- List FreeRTOS hardware platforms.

To grant your IAM user the required permissions, create an OTA user policy and then attach it to your IAM user. For more information, see IAM Policies.

To create an OTA user policy

1. Open the https://console.aws.amazon.com/iam/ console.
2. In the navigation pane, choose **Users**.
3. Choose your IAM user from the list.
4. Choose **Add permissions**.
5. Choose **Attach existing policies directly**.
6. Choose **Create policy**.
7. Choose the **JSON** tab, and copy and paste the following policy document into the policy editor:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:ListAllMyBuckets",
                "s3:CreateBucket",
                "s3:PutBucketVersioning",
                "s3:GetBucketLocation",
                "s3:GetObjectVersion",
                "acm:ImportCertificate",
                "acm:ListCertificates",
                "iot:*",
                "iam:ListRoles",
                "freertos:ListHardwarePlatforms",
                "freertos:DescribeHardwarePlatform"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::example-bucket/*"
        },
        {
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::your-account-id:role/role-name"
        }
    ]
}
```

   Replace *example-bucket* with the name of the Amazon S3 bucket where your OTA update firmware image is stored. Replace *your-account-id* with your AWS account ID. You can find your AWS account ID in the upper right of the console. When you enter your account ID, remove any dashes (-). Replace *role-name* with the name of the IAM service role you just created.
8. Choose **Review policy**.

9. Enter a name for your new OTA user policy, and then choose **Create policy**.

To attach the OTA user policy to your IAM user

1. In the IAM console, in the navigation pane, choose **Users**, and then choose your user.
2. Choose **Add permissions**.
3. Choose **Attach existing policies directly**.
4. Search for the OTA user policy you just created and select the check box next to it.
5. Choose **Next: Review**.
6. Choose **Add permissions**.

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 (p. 13)
- Creating a code-signing certificate for the Microchip Curiosity PIC32MZEF (p. 15)
- Creating a code-signing certificate for the Espressif ESP32 (p. 16)
- Creating a code-signing certificate for the Nordic nrf52840-dk (p. 17)
- Creating a code-signing certificate for the FreeRTOS Windows simulator (p. 18)
- Creating a code-signing certificate for custom hardware (p. 18)

Creating a code-signing certificate for the Texas Instruments CC3220SF-LAUNCHXL

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 Microchip Curiosity PIC32MZEF

The Microchip Curiosity PIC32MZEF 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 Espressif ESP32

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

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 `aws_ota_codesigner_certificate.h` file that is part of the FreeRTOS code you download later.

Grant access to code signing for AWS IoT

In production environments, you should digitally sign your firmware update to ensure the authenticity and integrity of the update. You can sign your update manually or you can use Code Signing for AWS IoT to sign your code. To use Code Signing for FreeRTOS, you must grant your IAM user account access to Code Signing for FreeRTOS.

To grant your IAM user account permissions for code signing for AWS IoT

1. Sign in to the https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Policies**.
3. Choose **Create Policy**.
4. On the **JSON** tab, copy and paste the following JSON document into the policy editor. This policy allows the IAM user access to all code-signing operations.

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

5. Choose **Review policy**.
6. Enter a policy name and description, and then choose **Create policy**.
7. In the navigation pane, choose **Users**.
8. Choose your IAM user account.
9. On the **Permissions** tab, choose **Add permissions**.
10. Choose **Attach existing policies directly**, and then select the check box next to the code-signing policy you just created.
11. Choose **Next: Review**.
12. Choose **Add permissions**.

Download FreeRTOS with the OTA library

You can download FreeRTOS from the FreeRTOS console (p. 65), or you can clone or download FreeRTOS from GitHub. See the README.md file for instructions.

To include the OTA library in the FreeRTOS configuration that you download from the console, you can customize a predefined configuration, or you can create a configuration for a platform that supports OTA functionality. On the **Configure FreeRTOS Software** configuration properties page, under **Libraries**, choose **OTA Updates**. Under **Demo projects**, you can choose to enable the OTA demo. You can also enable the demo manually later on.

For information about setting up and running the OTA demo application, see Over-the-air updates demo application (p. 257).

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 (MQTT, OTA Agent, 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 (p. 22) for more information.

By default, the configuration of the OTA protocols in `aws_ota_agent_config.h` is to use the MQTT protocol:

```
/**
 * @brief The protocol selected for OTA control operations.
 * This configuration parameter sets the default protocol for all the OTA control
 * operations like requesting OTA job, updating the job status etc.
 *
 * Note - Only MQTT is supported at this time for control operations.
 */
#define configENABLED_CONTROL_PROTOCOL       ( OTA_CONTROL_OVER_MQTT )
/**
 * @brief The protocol selected for OTA data operations.
 * This configuration parameter sets the protocols selected for the data operations
 * like requesting file blocks from the service.
 *
 * Note - Both MQTT and HTTP are supported for data transfer. This configuration parameter
 * can be set to the following -
 * Enable data over MQTT - ( OTA_DATA_OVER_MQTT )
 * Enable data over HTTP - ( OTA_DATA_OVER_HTTP)
 * Enable data over both MQTT & HTTP ( OTA_DATA_OVER_MQTT | OTA_DATA_OVER_HTTP )
 */
#define configENABLED_DATA_PROTOCOLS         ( OTA_DATA_OVER_MQTT )
 /**
  * @brief The preferred protocol selected for OTA data operations.
  *
  * Primary data protocol will be the protocol used for downloading files if more than
```

```
  * one protocol is selected while creating OTA job. Default primary data protocol is MQTT
  * and the following update here switches to HTTP as primary.
  *
  * Note - use OTA_DATA_OVER_HTTP for HTTP as primary data protocol.
  */
#define configOTA_PRIMARY_DATA_PROTOCOL     ( OTA_DATA_OVER_MQTT )
```

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.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:partition:iot:region:account:client/
${iot:Connection.Thing.ThingName}"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
                "arn:partition:iot:region:account:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/streams/*",
                "arn:partition:iot:region:account:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
                "arn:partition:iot:region:account:topic/$aws/things/
${iot:Connection.Thing.ThingName}/streams/*",
                "arn:partition:iot:region:account:topic/$aws/things/
${iot:Connection.Thing.ThingName}/jobs/*"
            ]
        }
    ]
}
```

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 MQTT library is still required because other interactions with AWS IoT Core use the MQTT 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 (MQTT, 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\aws_ota_agent_config.h` file.

```
/**
 * @brief The protocol selected for OTA control operations.
 * This configuration parameter sets the default protocol for all the OTA control
 * operations like requesting an OTA job, updating the job status, and so on.
 *
 * Note - Only MQTT is supported at this time for control operations.
 */
#define configENABLED_CONTROL_PROTOCOL       ( OTA_CONTROL_OVER_MQTT )
/**
 * @brief The protocol selected for OTA data operations.
 * This configuration parameter sets the protocols selected for the data operations
 * like requesting file blocks from the service.
 *
 * Note - Both MQTT and HTTP are supported for data transfer. This configuration parameter
 * can be set to the following -
 * Enable data over MQTT - ( OTA_DATA_OVER_MQTT )
 * Enable data over HTTP - ( OTA_DATA_OVER_HTTP)
```

```
 * Enable data over both MQTT & HTTP ( OTA_DATA_OVER_MQTT | OTA_DATA_OVER_HTTP )
 */
#define configENABLED_DATA_PROTOCOLS         ( OTA_DATA_OVER_MQTT )
 /**
  * @brief The preferred protocol selected for OTA data operations.
  *
  * Primary data protocol will be the protocol used for downloading files if more than
  * one protocol is selected while creating OTA job. Default primary data protocol is MQTT
  * and the following update here switches to HTTP as primary.
  *
  * Note - use OTA_DATA_OVER_HTTP for HTTP as primary data protocol.
  */
#define configOTA_PRIMARY_DATA_PROTOCOL     ( OTA_DATA_OVER_MQTT )
```

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.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource": "arn:partition:iot:region:account:client/
${iot:Connection.Thing.ThingName}"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Subscribe",
            "Resource": [
                "arn:partition:iot:region:account:topicfilter/$aws/things/
${iot:Connection.Thing.ThingName}/jobs/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish",
                "iot:Receive"
            ],
            "Resource": [
```

```
                "arn:partition:iot:region:account:topic/$aws/things/
${iot:Connection.Thing.ThingName}/jobs/*"
            ]
        }
    ]
}
```

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 (p. 26)

- Install the initial version of firmware on the Microchip Curiosity PIC32MZEF (p. 28)
- Install the initial version of firmware on the Espressif ESP32 (p. 30)
- Install the initial version of firmware on the Nordic nRF52840 DK (p. 33)
- Initial firmware on the Windows simulator (p. 33)
- Install the initial version of firmware on a custom board (p. 33)

Install the initial version of firmware on the Texas Instruments CC3220SF-LAUNCHXL

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 (p. 260).

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 Microchip Curiosity PIC32MZEF

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 Microchip Curiosity PIC32MZEF (p. 262).

To burn the demo application onto your board

1. Rebuild the `aws_demos` project and make sure it compiles without errors.
2. On the tool bar, choose  .
3. After the programming process is complete, disconnect the ICD 4 debugger and reset the board. Reconnect your terminal emulator to make sure the output is the same as when you debugged the demo with MPLAB X IDE.

   The terminal should display output similar to the following.

```
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 following procedure creates a unified hex file or factory image that consists of a reference bootloader and an application with a cryptographic signature. The bootloader verifies the cryptographic signature of the application on boot and supports OTA updates.

To build and flash a factory image

1. Make sure you have the SRecord tools installed from Source Forge. Verify that the directory that contains the `srec_cat` and `srec_info` programs is in your system path.
2. Update the OTA sequence number and application version for the factory image.
3. Build the `aws_demos` project.
4. Run the `factory_image_generator.py` script to generate the factory image.

```
factory_image_generator.py -b mplab.production.bin -p MCHP-Curiosity-PIC32MZEF -k
 private_key.pem  -x aws_bootloader.X.production.hex
```

This command takes the following parameters:

- `mplab.production.bin`: The application binary.
- `MCHP-Curiosity-PIC32MZEF`: The platform name.
- `private_key.pem`: The code-signing private key.
- `aws_bootloader.X.production.hex`: The bootloader hex file.

When you build the`aws_demos` project, the application binary image and bootloader hex file are built as part of the process. Each project under the `vendors/microchip/boards/curiosity_pic32mzef/aws_demos/` directory contains a `dist/pic32mz_ef_curiosity/production/` directory that contains these files. The generated unified hex file is named `mplab.production.factory.unified.hex`.

5. Use the MPLab IPE tool to program the generated hex file onto the device.
6. You can check that your factory image works by watching the board's UART output as the image is uploaded. If everything is set up correctly, you should see the image boot successfully:

```
[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] Valid magic code at: 0xbd000000
[prvValidateImage] Valid image flags: 0xfc at: 0xbd000000
[prvValidateImage] Addresses are valid.
[prvValidateImage] Crypto signature is valid.
[...]
[prvBOOT_ValidateImages] Booting image with sequence number 1 at 0xbd000000
```

7. If your certificates are incorrectly configured or if an OTA image is not properly signed, you might see messages like the following before the chip's bootloader erases the invalid update. Check that your code-signing certificates are consistent and review the previous steps carefully.

```
[prvValidateImage] Validating image at Bank : 0
[prvValidateImage] Valid magic code at: 0xbd000000
[prvValidateImage] Valid image flags: 0xfc at: 0xbd000000
[prvValidateImage] Addresses are valid.
[prvValidateImage] Crypto signature is not valid.
[prvBOOT_ValidateImages] Validation failed for image at 0xbd000000
[BOOT_FLASH_EraseBank] Bank erased at : 0xbd000000
```

Install the initial version of firmware on the Espressif ESP32

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_MQTT_DEMO_ENABLED`, and define `CONFIG_OTA_UPDATE_DEMO_ENABLED`.

2. Copy your SHA-256/ECDSA PEM-formatted code-signing certificate that you generated in the OTA update prerequisites (p. 9) to `demos/include/aws_ota_codesigner_certificate.h`. It should be formatted in following way.

```
static const char signingcredentialSIGNING_CERTIFICATE_PEM[] = "-----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

This guide is written with the assumption that you have already performed the steps in Getting started with the Nordic nRF52840-DK (p. 154) 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_MQTT_DEMO_ENABLED` with `#define democonfigOTA_UPDATE_DEMO_ENABLED`.
3. With the OTA Update demo selected, follow the same steps outlined in Getting started with the Nordic nRF52840-DK (p. 154) 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 (p. 236).

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. Open `demos/include/aws_application_version.h` and increment the `APP_VERSION_BUILD` token value.
3. If you are using the Microchip Curiosity PIC32MZEF, increment the OTA sequence number in `vendors/microchip/boards/curiosity_pic32mzef/bootloader/bootloader/utility/user-config/ota-descriptor.config`. The OTA sequence number should be incremented for every new OTA image generated.
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 (p. 9). 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`
- Microchip Curiosity PIC32MZEF: `vendors/microchip/boards/curiosity_pic32mzef/aws_demos/mplab/dist/pic32mz_ef_curiosity/production/mplab.production.ota.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, choose **Manage**, and then choose **Jobs**.
2. Choose **Create**.
3. Under **Create a FreeRTOS Over-the-Air (OTA) update job**, choose **Create OTA update job**.
4. You can deploy an OTA update to a single device or a group of devices. Under **Select devices to update**, choose **Select**. To update a single device, choose the **Things** tab. To update a group of devices, choose the **Thing Groups** tab.
5. If you are updating a single device, select the check box next to the IoT thing associated with your device. If you are updating a group of devices, select the check box next to the thing group associated with your devices. Choose **Next**.
6. Under **Select the protocol for firmware image transfer**, choose **HTTP**, **MQTT**, or choose both to allow each device to determine the protocol to use.
7. Under **Select and sign your firmware image**, choose **Sign a new firmware image for me**.
8. Under **Code signing profile**, choose **Create**.
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 (p. 37).

b. Under **Code signing certificate**, choose **Select** to select a previously imported certificate or **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. The certificate's location varies by platform. It should be the location where you put the code-signing certificate when you followed the instructions in Installing the initial firmware (p. 25).

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.

10. Under **Select your firmware image in S3 or upload it**, choose **Select**. 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 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.

Important
On the Texas Instruments CC3220SF-LAUNCHXL, due to security restrictions, the firmware image path name must be `/sys/mcuflashimg.bin`.

12. Under **IAM role for OTA update job**, choose a role according to the instructions in Create an OTA Update service role (p. 10).
13. Choose **Next**.
14. Enter an ID and description for your OTA update job.
15. Under **Job type**, choose **Your job will complete after deploying to the selected devices/groups (snapshot)**.
16. Choose any appropriate optional configurations for your job (**Job executions rollout**, **Job abort**, **Job executions timeout**, and **Tags**).
17. 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. This path name might vary by platform.
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.
- If you see `"Error: You have exceeded the limit for the number of streams in your AWS account."`, then see Stream limit exceeded for your AWS account (p. 64).

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 (p. 257).

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 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 for the signed firmware in an S3 bucket. 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** 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** 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 (p. 10) 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**

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 ...
```

Note
If you see `"Error: You have exceeded the limit for the number of streams in your AWS account."`, then see Stream limit exceeded for your AWS account (p. 64).

Creating an OTA update

Use the **create-ota-update** 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",
    "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.
`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.

Name	Type	Description
		Valid values: CONTINUOUS | SNAPSHOT. enum: CONTINUOUS | SNAPSHOT
`awsJobExecutionsRolloutConfig`		Configuration for the rollout of OTA updates.
`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.
`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 abort takes place.
`abortCriteriaList`	list	The list of criteria that determine when and how to abort the job.

Name	Type	Description
`failureType`	string	The type of job execution failures that can initiate a job abort. enum: FAILED | REJECTED | TIMED_OUT | ALL
`action`	string	The type of job action to take to initiate the job abort. enum: CANCEL
`minNumberOfExecutedThings`	integer (min:1)	The minimum number of things which must receive job execution notifications before the job can be aborted.
`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.
`fileVersion`	string	The file version.
`fileLocation`		The location of the updated firmware.
`stream`		The stream that contains the OTA update.

Name	Type	Description
`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.
`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.
`Value`	string (max:256 min:1)	The tag's value.

Output

```
{
  "otaUpdateId": "string",
  "awsIotJobId": "string",
  "otaUpdateArn": "string",
  "awsIotJobArn": "string",
  "otaUpdateStatus": "string"
}
```

CLI output fields

Name	Type	Description
`otaUpdateId`	string	The OTA update ID.

Name	Type	Description
	(max:128 min:1)	
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",
    "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** CLI command that uses an inline file to provide custom code-signing material.

```
[
  {
    "fileName": "firmware.bin",
    "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** 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",
    "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** 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",
    "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** CLI command that allows FreeRTOS OTA to create a stream with an existing code-signing job ID.

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileVersion": "1",
    "codeSigning":{
      "awsSignerJobId": "your_signer_job_id"
    }
  }
]
```

The following is an example of a JSON file passed into the **create-ota-update** 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",
    "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"
      }
    }
  }
]
```

Note
If you see `"Error: You have exceeded the limit for the number of streams in your AWS account."`, then see Stream limit exceeded for your AWS account (p. 64).

Listing OTA updates

You can use the **list-ota-updates** 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** 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:us-west-2:123456789012:otaupdate/ota-update-001",
        "creationDate": 1575414146.286,
        "lastModifiedDate": 1575414149.091,
        "targets": [
            "arn:aws:iot:us-west-2:123456789012:thing/myDevice"
        ],
        "protocols": [ "HTTP" ],
        "awsJobExecutionsRolloutConfig": {
            "maximumPerMinute": 0
        },
        "awsJobPresignedUrlConfig": {
            "expiresInSec": 1800
        },
        "targetSelection": "SNAPSHOT",
        "otaUpdateFiles": [
            {
                "fileName": "my_firmware.bin",
                "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:us-west-2:123456789012: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** 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** 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** 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 (p. 257).

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, optionally, creating a custom callback function for responding to the OTA completion event messages.

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 file. If you choose HTTP, the OTA Agent handles the HTTP connections.

Simple OTA demo using MQTT

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 completion callback and print out some statistics once per second. For brevity, we leave out some details from this demo.

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.

```
void vRunOTAUpdateDemo( const IotNetworkInterface_t * pNetworkInterface,
                        void * pNetworkCredentialInfo )
{
    IotMqttConnectInfo_t xConnectInfo = IOT_MQTT_CONNECT_INFO_INITIALIZER;
    OTA_State_t eState;
    OTA_ConnectionContext_t xOTAConnectionCtx = { 0 };

    configPRINTF( ( "OTA demo version %u.%u.%u\r\n",
                    xAppFirmwareVersion.u.x.ucMajor,
                    xAppFirmwareVersion.u.x.ucMinor,
                    xAppFirmwareVersion.u.x.usBuild ) );
    configPRINTF( ( "Creating MQTT Client...\r\n" ) );

    /* Create the MQTT Client. */

    for( ; ; )
    {
        xNetworkConnected = prxCreateNetworkConnection();

        if( xNetworkConnected )
        {
            configPRINTF( ( "Connecting to broker...\r\n" ) );
            memset( &xConnectInfo, 0, sizeof( xConnectInfo ) );

            if( xConnection.ulNetworkType == AWSIOT_NETWORK_TYPE_BLE )
            {
                xConnectInfo.awsIotMqttMode = false;
                xConnectInfo.keepAliveSeconds = 0;
            }
```

```
            else
            {
                xConnectInfo.awsIotMqttMode = true;
                xConnectInfo.keepAliveSeconds = otaDemoKEEPALIVE_SECONDS;
            }

            xConnectInfo.cleanSession = true;
            xConnectInfo.clientIdentifierLength = ( uint16_t )
strlen( clientcredentialIOT_THING_NAME );
            xConnectInfo.pClientIdentifier = clientcredentialIOT_THING_NAME;

            /* Connect to the broker. */
            if( IotMqtt_Connect( &( xConnection.xNetworkInfo ),
                                 &xConnectInfo,
                                 otaDemoCONN_TIMEOUT_MS, &( xConnection.xMqttConnection ) )
== IOT_MQTT_SUCCESS )
            {
                configPRINTF( ( "Connected to broker.\r\n" ) );
                xOTAConnectionCtx.pvControlClient = xConnection.xMqttConnection;
                xOTAConnectionCtx.pxNetworkInterface = ( void * ) pNetworkInterface;
                xOTAConnectionCtx.pvNetworkCredentials = pNetworkCredentialInfo;

                OTA_AgentInit( ( void * ) ( &xOTAConnectionCtx ), ( const uint8_t * )
( clientcredentialIOT_THING_NAME ), App_OTACompleteCallback, ( TickType_t ) ~0 );

                while( ( eState = OTA_GetAgentState() ) != eOTA_AgentState_Stopped )
                {
                    /* Wait forever for OTA traffic but allow other tasks to run and output
statistics only once per second. */
                    vTaskDelay( myappONE_SECOND_DELAY_IN_TICKS );
                    configPRINTF( ( "State: %s  Received: %u   Queued: %u   Processed: %u
Dropped: %u\r\n", pcStateStr[ eState ],
                                    OTA_GetPacketsReceived(), OTA_GetPacketsQueued(),
OTA_GetPacketsProcessed(), OTA_GetPacketsDropped() ) );
                }

                IotMqtt_Disconnect( xConnection.xMqttConnection, false );
            }
            else
            {
                configPRINTF( ( "ERROR:  MQTT_AGENT_Connect() Failed.\r\n" ) );
            }

            vMqttDemoDeleteNetworkConnection( &xConnection );

            /* After failure to connect or a disconnect, wait an arbitrary one second
before retry. */
            vTaskDelay( myappONE_SECOND_DELAY_IN_TICKS );
        }
        else
        {
            configPRINTF( ( "Failed to create MQTT client.\r\n" ) );
        }
    }
}
```

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 allowing an OTA update job and output statistics once a second.
- If the Agent stops, wait one second and try connecting again.

Using a custom callback for OTA completion events

The previous example used the built-in callback handler for OTA completion events by specifying `NULL` for the third parameter to the `OTA_AgentInit` API. If you want to implement custom handling of the completion events, you must pass the function address of your callback handler to the `OTA_AgentInit` API. During the OTA process, the 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.

```
/**
* @brief OTA Job callback events.
*
* After an OTA update image is received and authenticated, the Agent calls the user
* callback (set with the OTA_AgentInit API) with the value eOTA_JobEvent_Activate 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 eOTA_JobEvent_StartTest, 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 eOTA_JobEvent_Fail instead to allow the user to log the failure and take
* any action deemed appropriate by the user code.
*
*/
typedef enum {
    eOTA_JobEvent_Activate,  /*! OTA receive is authenticated and ready to activate. */
    eOTA_JobEvent_Fail,      /*! OTA receive failed. Unable to use this update. */
    eOTA_JobEvent_StartTest  /*! OTA job is now in self test, perform user tests. */
} OTA_JobEvent_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:

`eOTA_JobEvent_Activate event`

When this event is received by the callback handler, 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.

`eOTA_JobEvent_Fail event`

When this event is received by the callback handler, 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.

`eOTA_JobEvent_StartTest event`

The self-test phase is meant to allow newly updated firmware to execute and test itself before determining that it is properly functioning and commit it 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 `eOTA_JobEvent_StartTest` 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( eOTA_ImageState_Accepted )` function.

If your device has no special hardware or mechanisms that need to be tested, you can use the default callback handler. Upon receipt of the `eOTA_JobEvent_Activate` event, the default handler resets the device immediately.

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 (p. 58)
- Log AWS IoT OTA API calls with AWS CloudTrail (p. 61)
- Get OTA failure codes with the AWS CLI (p. 63)
- Troubleshoot OTA updates of multiple devices (p. 64)

- Troubleshoot OTA updates with the Texas Instruments CC3220SF Launchpad (p. 64)
- Stream limit exceeded for your AWS account (p. 64)

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 Streaming service.
- `DescribeStream`: A request for information about a stream was sent to the AWS IoT Streaming service.

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 streaming service:

```
{
    "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 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
```

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 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`.

Stream limit exceeded for your AWS account

Although there is no charge to use FreeRTOS, creating a stream might incur charges to your account. Because the OTA Cloud service accesses S3 object metadata in your AWS account on your behalf, this might generate a cost on your bill. For more information, see Amazon S3 pricing.

If you see "`Error: You have exceeded the limit for the number of streams in your AWS account.`", you can clean up the unused streams in your account instead of requesting a limit increase.

To clean up unused streams, use the following commands.

For a stream created by the OTA Update Manager Service:

```
aws iot delete-ota-update -ota-update-id value --delete-stream
```

For more details, see delete-ota-update.

For a stream that you created using the AWS CLI or SDK:

```
aws iot delete-stream -stream-id value
```

For more details, see delete-stream.

Note
You can use the `list-ota-updates` or `list-streams` commands to find the OTA update ID or stream ID.

Downloading FreeRTOS source code

You can download versions of FreeRTOS that are configured for FreeRTOS-qualified platforms from the FreeRTOS console. For a list of qualified platforms, see FreeRTOS-qualified hardware platforms (p. 68) or the FreeRTOS Partners website.

You can also clone or download FreeRTOS from GitHub. See the README.md file for instructions.

FreeRTOS console

From the FreeRTOS console, you can configure and download a package that contains everything you need to write an application for your microcontroller-based devices:

- The FreeRTOS kernel.
- FreeRTOS libraries.
- Platform support libraries.
- Hardware drivers.

For more information, see FreeRTOS console (p. 65).

FreeRTOS console

In the FreeRTOS console, you can download a package with a predefined configuration, or you can create your own configuration by selecting your hardware platform and the libraries required for your application. These configurations are saved in AWS and are available for download at any time.

Predefined FreeRTOS configurations

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

You can also customize a predefined configuration if you want to change the FreeRTOS version, hardware platform, or libraries of the configuration. Customizing a predefined configuration creates a new custom configuration and does not overwrite the predefined configuration in the FreeRTOS console.

To create a custom configuration from a predefined configuration

1. Browse to the FreeRTOS console.

2. In the navigation pane, choose **Software**.
3. Under **FreeRTOS Device Software**, choose **Configure download**.
4. Choose the ellipsis (**...**) next to the predefined configuration that you want to customize, and then choose **Customize**.
5. On the **Configure FreeRTOS Software** page, choose the FreeRTOS version, hardware platform, and libraries, and give the new configuration a name and a description.
6. At the bottom of the page, choose **Create and download**.

Custom FreeRTOS configurations

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

To create a custom configuration

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

   Note
   Do not use any personally identifiable information in your custom configuration name.
8. In **Description**, enter a description for your custom configuration.
9. At the bottom of the page, choose **Create and download**.

To edit a custom configuration

1. Browse to the FreeRTOS console.
2. In the navigation pane, choose **Software**.
3. Under **FreeRTOS Device Software**, choose **Configure download**.
4. Choose the ellipsis (**...**) next to the configuration you want to edit, and then choose **Edit**.
5. On the **Configure FreeRTOS Software** page, you can change your configuration's FreeRTOS version, hardware platform, libraries, and description.
6. At the bottom of the page, choose **Save and download**.

To delete a custom configuration

1. Browse to the FreeRTOS console.
2. In the navigation pane, choose **Software**.
3. Under **FreeRTOS Device Software**, choose **Configure download**.
4. Choose the ellipsis (**...**) next to the configuration you want to delete, and then choose **Delete**.

Quick connect workflow

The FreeRTOS console also includes the Quick Connect workflow option for all boards with predefined configurations. The Quick Connect workflow helps you configure and run FreeRTOS demo applications for AWS IoT and AWS IoT Greengrass. To get started, choose the **Predefined configurations** tab, find your board, choose **Quick connect**, and then follow the Quick Connect workflow steps.

Tagging configurations

You can apply tags to FreeRTOS configurations when you create or edit a configuration in the console. To apply tags to a configuration, navigate to the console. Under **Tags**, enter the name and value for the tag.

You can use tags to manage access permissions to configurations with IAM policies. For information, see Using tags with IAM policies (p. 67).

For more information about using tags to manage AWS IoT resources, see Using Tags with IAM Policies in the *AWS IoT Developer Guide*.

Using tags with IAM policies

You can use the FreeRTOS console to apply tag-based, resource-level permissions in the IAM policies that you use for operations. This gives you better control over which configurations a user can create, modify, or use. For more information about using tagging and IAM policies for AWS IoT, see Using Tags with IAM Policies in the *AWS IoT Developer Guide*.

In the IAM policy definition, use the `Condition` element (also called the `Condition` block) with the following condition context keys and values to control user access (permissions) based on a resource's tags:

- Use `aws:ResourceTag/`*`tag-key`*`: `*`tag-value`* to allow or deny user actions on FreeRTOS configurations with specific tags.
- Use `aws:RequestTag/`*`tag-key`*`: `*`tag-value`* to require that a specific tag be used (or not used) when creating or modifying a configuration in the FreeRTOS console.
- Use `aws:TagKeys: [`*`tag-key`*`, ...]` to require that a specific set of tag keys be used (or not used) when creating or modifying a configuration in the FreeRTOS console.

For more information, see Controlling Access Using Tags in the *AWS Identity and Access Management User Guide*. For detailed syntax, descriptions, and examples of the elements, variables, and evaluation logic of JSON policies in IAM, see the IAM JSON Policy Reference.

The following example policy applies two tag-based restrictions. An IAM user restricted by this policy:

- Cannot give a resource the tag `env=prod` (in the example, see the line `"aws:RequestTag/env" : "prod"`
- Cannot modify or access a resource that has an existing tag `env=prod` (in the example, see the line `"aws:ResourceTag/env" : "prod"`).

```
{
    "Version" : "2012-10-17",
    "Statement" : [
        {
          "Effect" : "Deny",
          "Action" : "freertos:*",
          "Resource" : "*",
          "Condition" : {
```

```
            "StringEquals" : {
              "aws:RequestTag/env" : "prod"
            }
          }
        },
        {
          "Effect" : "Deny",
          "Action" : "freertos:*",
          "Resource" : "*",
          "Condition" : {
            "StringEquals" : {
              "aws:ResourceTag/env" : "prod"
            }
          }
        },
        {
          "Effect": "Allow",
          "Action": [
            "iot:*"
          ],
          "Resource": "*"
        }
    ]
}
```

You can also specify multiple tag values for a given tag key by enclosing them in a list, like this:

```
{
    ...
    "StringEquals" : {
        "aws:ResourceTag/env" : ["dev", "test"]
    }
}
```

Note
If you allow or deny users access to resources based on tags, you must consider explicitly denying users the ability to add those tags to or remove them from the same resources. Otherwise, it's possible for a user to circumvent your restrictions and gain access to a resource by modifying its tags.

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
- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- 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

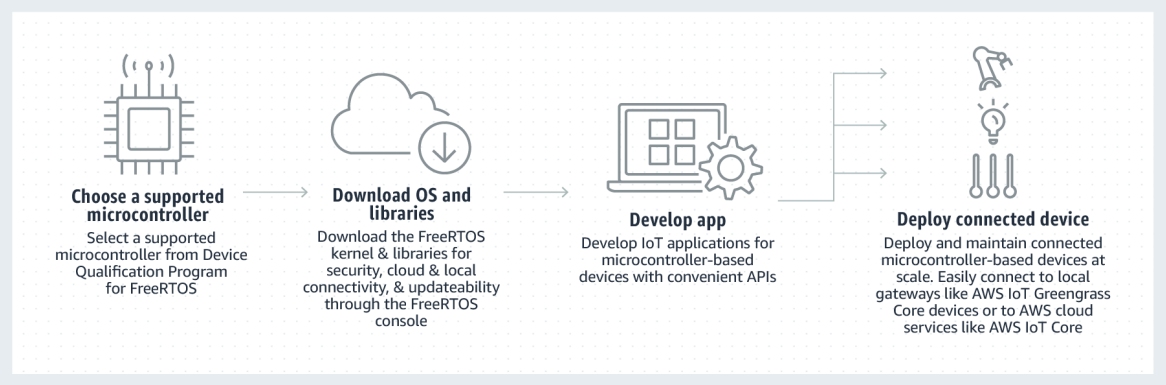
- NuMaker-IoT-M487
- NXP LPC54018 IoT Module
- OPTIGA Trust X Security Solution
- Renesas RX65N RSK IoT Module
- STMicroelectronicsSTM32L4 Discovery Kit IoT Node
- Texas Instruments CC3220SF-LAUNCHXL
- Microsoft Windows 7 or later, with at least a dual core and a hard-wired Ethernet connection
- Xilinx 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.

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.



AWS IoT Device SDK for Embedded C

The AWS IoT Device SDK for Embedded C is a collection of C source files that can be used in embedded applications to securely connect to the AWS IoT platform. The SDK includes transport clients, TLS implementations, and usage examples. It also includes libraries that interact with AWS IoT services on the AWS Cloud.

The AWS IoT Device SDK for Embedded C includes the following libraries. These libraries have the same APIs as their corresponding FreeRTOS libraries.

- AWS IoT Device Defender
- MQTT
- AWS IoT Device Shadow
- Common Libraries
  - Linear Containers
  - Logging

- Static Memory
- Task Pool

The SDK is distributed as source code that is intended to be built into customer firmware along with application code, other libraries, and RTOS. For more information, see the AWS IoT Device SDK for Embedded C GitHub.

For instructions on porting the SDK source code to your environment, see the AWS IoT Device SDK for Embedded C Porting Guide.

For an API reference, see the AWS IoT Device SDK C API Reference.

Additional resources

These resources might be helpful to you.

- For questions about FreeRTOS for the FreeRTOS engineering team, you can open an issue on the FreeRTOS GitHub page.
- For technical questions about FreeRTOS visit the FreeRTOS Community Forums.
- For technical support for AWS, visit the AWS Support Center.
- For questions about AWS billing, account services, events, abuse, or other issues with AWS, visit the Contact Us page.

Getting Started with FreeRTOS

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.

FreeRTOS demo application

The demo application in this tutorial is the Hello World MQTT demo defined in the `/demos/mqtt/iot_demo_mqtt.c` file. It uses the FreeRTOS MQTT library (p. 218) 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 Hello World MQTT demo is enabled by default.

For more information about the demo applications included with FreeRTOS, see FreeRTOS Demos (p. 236).

First steps

To get started, see First steps (p. 72).

Board-specific getting started guides

After you complete the First steps (p. 72), you can 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 (p. 86).

Troubleshooting

For help troubleshooting any issues that you encounter while getting started, see Troubleshooting getting started (p. 79). For board-specific troubleshooting tips, see the Getting Started guide for your board in Board-specific getting started guides (p. 86).

Developing FreeRTOS applications

You can 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.

You can also use third-party code editors and debuggers to develop applications, and CMake to build and run the source code. For more information about using CMake as a build tool for FreeRTOS development, see Using CMake with FreeRTOS (p. 80).

First steps

To get started using FreeRTOS with AWS IoT, you need an AWS account, an IAM user with permission to access AWS IoT and FreeRTOS cloud services. You also need to 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 (p. 97).
> If you're using the Nordic nRF52840-DK, skip these steps and go to Getting started with the Nordic nRF52840-DK (p. 154).

1. Setting up your AWS account and permissions (p. 72)

   After you complete the instructions in Setting up your AWS account and permissions (p. 72), you can follow the **Quick Connect** workflow in the FreeRTOS console to quickly connect your board to the AWS Cloud. If you follow the **Quick Connect** workflow, you do not need to complete the remaining steps in this list. Note that configurations of FreeRTOS are currently not available on the FreeRTOS console for the following boards:

   - Cypress CYW943907AEVAL1F Development Kit
   - Cypress CYW954907AEVAL1F Development Kit
2. Registering your MCU board with AWS IoT (p. 73)
3. Downloading FreeRTOS (p. 75)
4. Configuring the FreeRTOS demos (p. 75)

Setting up your AWS account and permissions

To create an AWS account, see Create and Activate an AWS Account.

To add an IAM user to your AWS account, see IAM User Guide. To grant your IAM user account access to AWS IoT and FreeRTOS, attach the following IAM policies to your IAM user account:

- `AmazonFreeRTOSFullAccess`
- `AWSIoTFullAccess`

To attach the AmazonFreeRTOSFullAccess policy to your IAM user

1. Browse to the IAM console, and from the navigation pane, choose **Users**.
2. Enter your user name in the search text box, and then choose it from the list.
3. Choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. In the search box, enter **`AmazonFreeRTOSFullAccess`**, choose it from the list, and then choose **Next: Review**.
6. Choose **Add permissions**.

To attach the AWSIoTFullAccess policy to your IAM user

1. Browse to the IAM console, and from the navigation pane, choose **Users**.
2. Enter your user name in the search text box, and then choose it from the list.
3. Choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. In the search box, enter **`AWSIoTFullAccess`**, choose it from the list, and then choose **Next: Review**.
6. Choose **Add permissions**.

For more information about IAM and user accounts, see IAM User Guide.

For more information about policies, see IAM Permissions and Policies.

After you set up your AWS account and permissions, you can continue to Registering your MCU board with AWS IoT (p. 73) or to the **Quick Connect** workflow in the FreeRTOS console.

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 need the following:

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.

If you use the **Quick Connect** workflow in the FreeRTOS console, a policy, an AWS IoT thing, and a key and certificate are created for you. If you use the **Quick Connect** workflow, you can ignore the following procedures.

To register your board manually, follow the procedures below.

To create an AWS IoT policy

1. To create an IAM policy, you need to 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
   ```

   The output should look like this:

   ```
   {
       "endpointAddress": "xxxxxxxxxxxxxx.iot.us-west-2.amazonaws.com"
   ```

```
}
```

In this example, the region is `us-west-2`.

2. Browse to the AWS IoT console.
3. 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.

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Effect": "Allow",
        "Action": "iot:Connect",
        "Resource":"arn:aws:iot:aws-region:aws-account-id:*"
    },
    {
        "Effect": "Allow",
        "Action": "iot:Publish",
        "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
    },
    {
         "Effect": "Allow",
         "Action": "iot:Subscribe",
         "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
    },
    {
         "Effect": "Allow",
         "Action": "iot:Receive",
         "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
    }
    ]
}
```

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 (p. 75).

Downloading FreeRTOS

You can download FreeRTOS from the FreeRTOS console or from the FreeRTOS GitHub repository.

> **Note**
> If you're following the **Quick Connect** workflow in the FreeRTOS console, you can ignore the following procedure..

To download FreeRTOS from the FreeRTOS console

1. Sign in to the FreeRTOS console.
2. Under **Predefined configurations**, find **Connect to AWS IoT- *Platform***, and then choose **Download**.
3. Unzip the downloaded file to a directory, and copy the directory path.

   > **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.

> **Note**
> If you're getting started with the Cypress CYW954907AEVAL1F or CYW943907AEVAL1F development kits, you must download FreeRTOS from GitHub. See the README.md file for instructions. Configurations of FreeRTOS for these boards aren't currently available from the FreeRTOS console.

After you download FreeRTOS, you can continue to Configuring the FreeRTOS demos (p. 75).

Configuring the FreeRTOS demos

You need to edit some configuration files in your FreeRTOS directory before you can compile and run any demos on your board.

If you are following the **Quick Connect** workflow on the FreeRTOS console, follow the configuration instructions in the workflow on the console, and ignore these procedures.

To configure your AWS IoT endpoint

You need to 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 navigation pane, choose **Settings**.

   Your AWS IoT endpoint is displayed in **Endpoint**. It should look like *1234567890123*-ats.iot.*us-east-1*.amazonaws.com. Make a note of this endpoint.
3. In the navigation pane, choose **Manage**, and then choose **Things**.

   Your device should have an AWS IoT thing name. Make a note of this name.
4. 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 need to 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 needs 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 need 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 compile and run the FreeRTOS demo. The demo application that is used in the Getting Started tutorial is the Hello World MQTT demo, which is located at `demos/mqtt/aws_hello_world.c`.

Developer-mode key provisioning

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 (p. 75).

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 (p. 75), 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 Hello World MQTT demo as described in the guide for your board in Board-specific getting started guides (p. 86) . 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

Since the device has not yet been provisioned with a private key and client certificate, the demo will fail to authenticate to AWS IoT. However, the Hello World MQTT 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 MQTT Hello World demo again. This time, the connection to the AWS IoT MQTT broker will succeed.

Troubleshooting getting started

The following topics can help you troubleshoot issues that you encounter while getting started with FreeRTOS:

Topics

- General getting started troubleshooting tips (p. 79)
- Installing a terminal emulator (p. 79)

For board-specific troubleshooting, see the Getting Started with FreeRTOS (p. 71) guide for your board.

General getting started troubleshooting tips

- If no messages appear in the AWS IoT console after you run the Hello World demo project, 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.

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

Terminal Setting	Value
Data	8 bit
Parity	none
Stop	1 bit
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

You can use CMake to generate project build files from FreeRTOS application source code.

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 (p. 81)
- Developing FreeRTOS applications with third-party code editors and debugging tools (p. 81)
- Building FreeRTOS with CMake (p. 82)

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 (p. 82).

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) (p. 82) 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) (p. 82) and Generating build files (CMake GUI) (p. 83).
2. Invoke the native build system to make the project into an executable.

   For information about making FreeRTOS build files, see Building FreeRTOS from generated build files (p. 85).

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
```

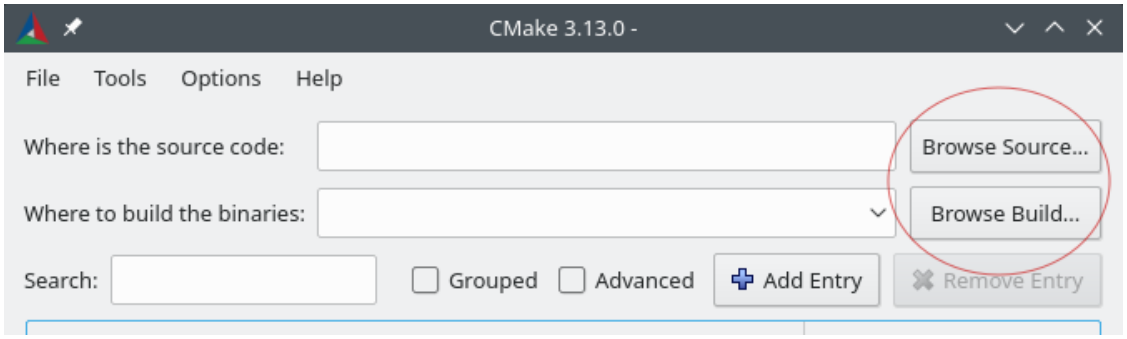
You can also set the `CMAKE_BUILD_TYPE` to `release` to add optimization flags to the compile options.

Generating build files (CMake GUI)

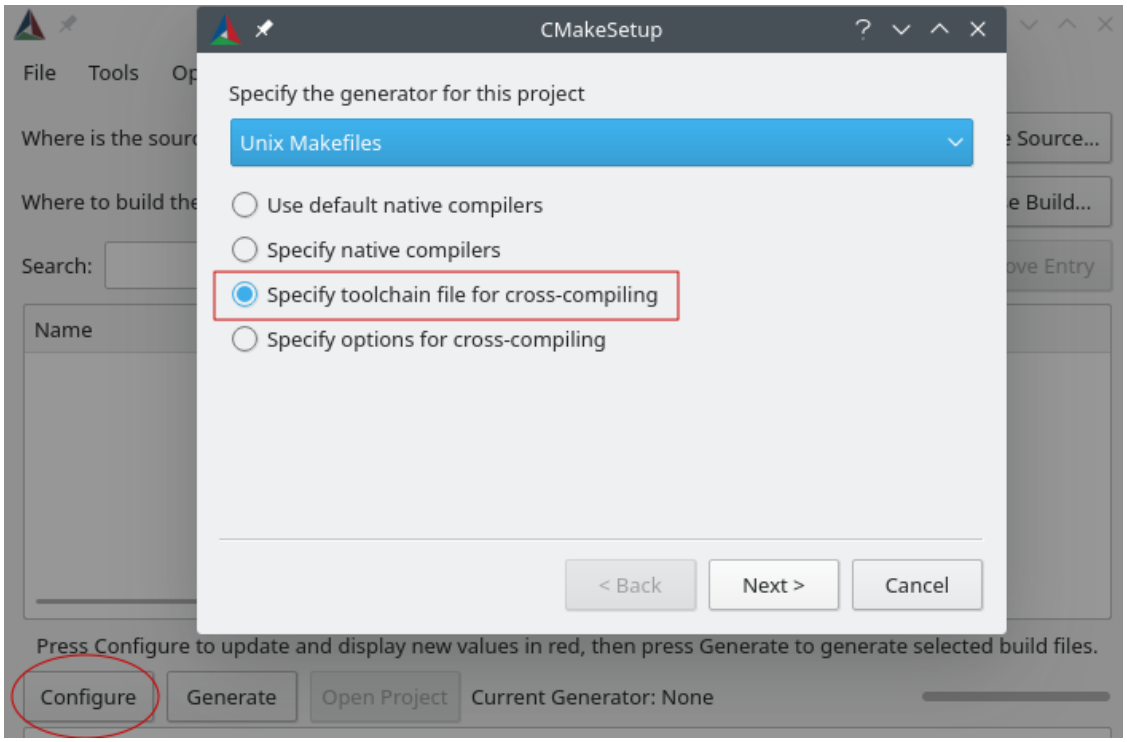
You can use the CMake GUI to generate FreeRTOS build files.

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.

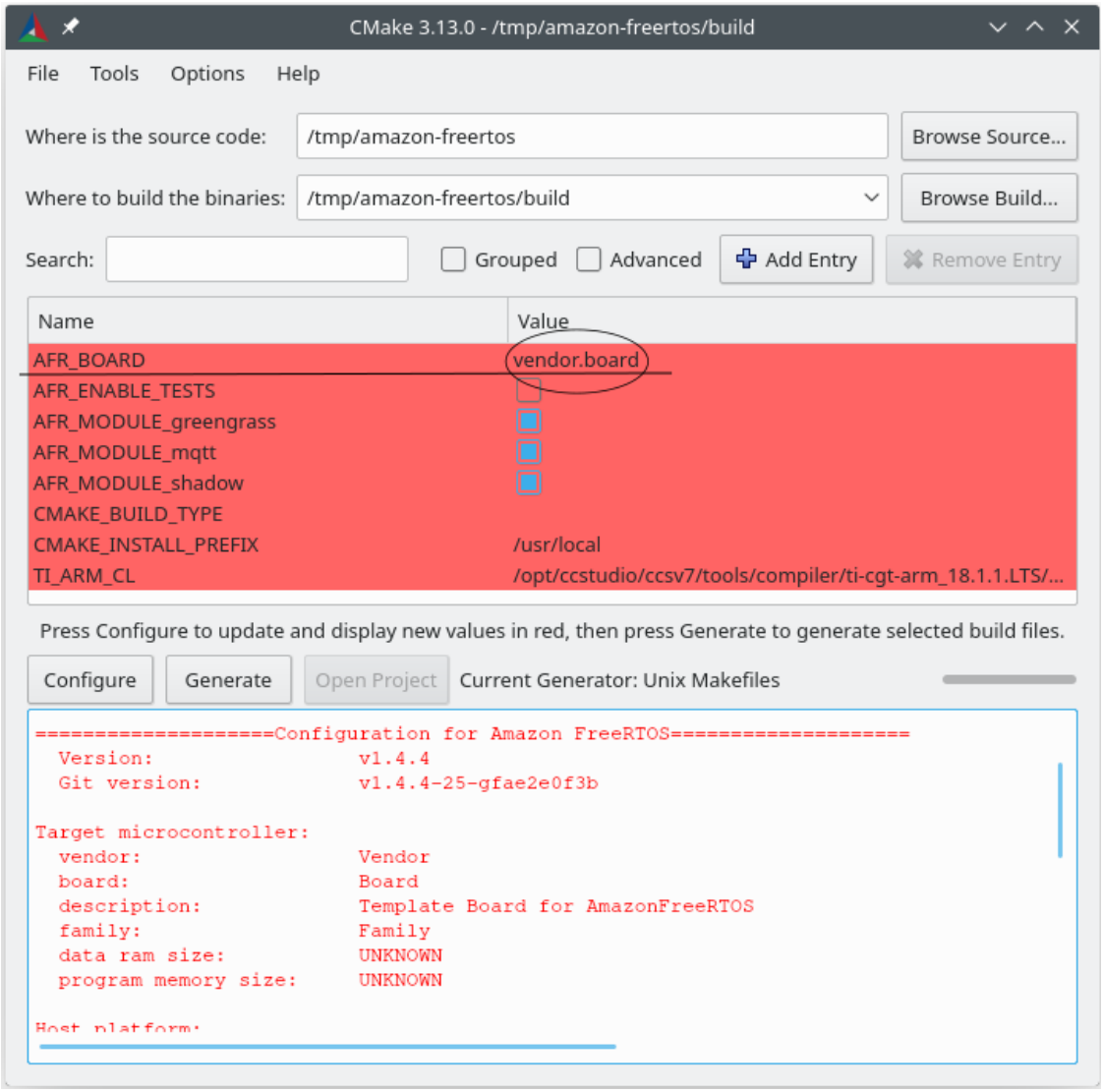
   Note
   If you are seeing the following error:

   ```
   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.

Board-specific getting started guides

After you complete the First steps (p. 72), see your board's guide for board-specific instructions on getting started with FreeRTOS:

- Getting started with the Cypress CYW943907AEVAL1F Development Kit (p. 87)
- Getting started with the Cypress CYW954907AEVAL1F Development Kit (p. 90)
- Getting started with the Infineon XMC4800 IoT Connectivity Kit (p. 117)
- Getting started with the Marvell MW320 AWS IoT Starter Kit (p. 126)
- Getting started with the Marvell MW322 AWS IoT Starter Kit (p. 136)
- Getting started with the MediaTek MT7697Hx development kit (p. 147)
- Getting started with the Microchip Curiosity PIC32MZ EF (p. 151)
- Getting started with the Nuvoton NuMaker-IoT-M487 (p. 158)
- Getting started with the NXP LPC54018 IoT Module (p. 164)
- Getting started with the Renesas Starter Kit+ for RX65N-2MB (p. 167)
- Getting started with the STMicroelectronics STM32L4 Discovery Kit IoT Node (p. 170)
- Getting started with the Texas Instruments CC3220SF-LAUNCHXL (p. 172)
- Getting started with the Windows Device Simulator (p. 176)
- Getting started with the Xilinx Avnet MicroZed Industrial IoT Kit (p. 178)

Note
You do not need to complete the First steps (p. 72) for the following self-contained Getting Started with FreeRTOS guides:

- Getting started with the Microchip ATECC608A Secure Element with Windows simulator (p. 93)
- Getting started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT (p. 97)
- Getting started with the Espressif ESP32-WROOM-32SE (preview) (p. 112)
- Getting started with the Infineon OPTIGA Trust X and XMC4800 IoT Connectivity Kit (p. 121)
- Getting started with the Nordic nRF52840-DK (p. 154)

Getting started with the Cypress CYW943907AEVAL1F Development Kit

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 MQTT Hello World 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 (p. 72) 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.
- As noted in Downloading FreeRTOS (p. 75), 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 (p. 79).

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.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 79).

Getting started with the Cypress CYW954907AEVAL1F Development Kit

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 MQTT Hello World 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 (p. 72) 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.
> - As noted in Downloading FreeRTOS (p. 75), 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 (p. 79).

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.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 79).

Getting started with the Microchip ATECC608A Secure Element with Windows simulator

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. 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.

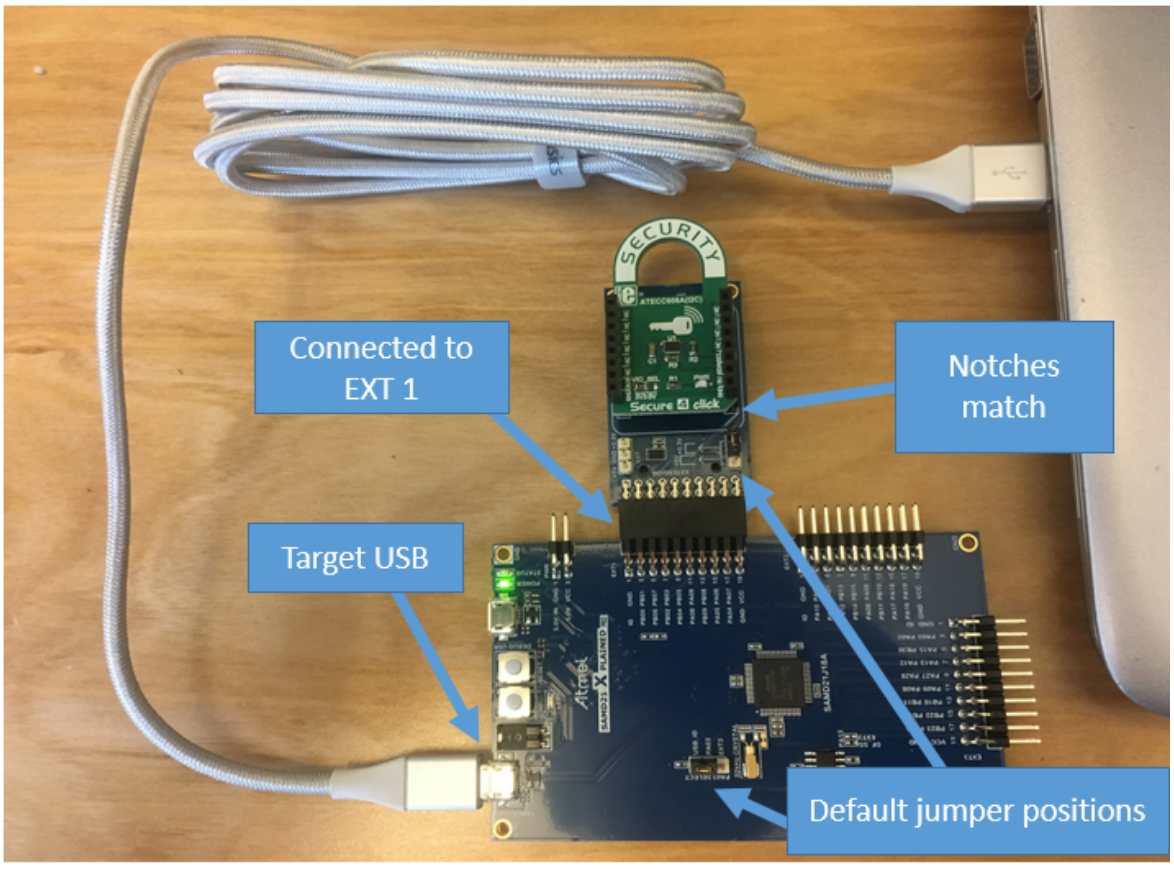
To set up the SAMD21 XPlained Pro board

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 Amtel 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

1. If you haven't already, create an AWS account. To add an IAM user to your AWS account, see IAM User Guide.

To grant your IAM user account access to AWS IoT and FreeRTOS, you attach the following IAM policies to your IAM user account in these steps:

- `AmazonFreeRTOSFullAccess`
- `AWSIoTFullAccess`

2. Attach the `AmazonFreeRTOSFullAccess` policy to your IAM user.

   a. Browse to the IAM console, and from the navigation pane, choose **Users**.

   b. Enter your user name in the search text box, and then choose it from the list.

   c. Choose **Add permissions**.

   d. Choose **Attach existing policies directly**.

   e. In the search box, enter **`AmazonFreeRTOSFullAccess`**, choose it from the list, and then choose **Next: Review**.

   f. Choose **Add permissions**.

3. Attach the `AWSIoTFullAccess` policy to your IAM user.

   a. Browse to the IAM console, and from the navigation pane, choose **Users**.

   b. Enter your user name in the search text box, and then choose it from the list.

   c. Choose **Add permissions**.

   d. Choose **Attach existing policies directly**.

   e. In the search box, enter **`AWSIoTFullAccess`**, choose it from the list, and then choose **Next: Review**.

   f. Choose **Add permissions**.

   For more information about IAM, see IAM Permissions and Policies in the IAM User Guide.

4. 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.

   4. From the *`freertos`* directory, check out the branch to use.

5. 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 (p. 177).

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 (p. 79).

Getting started with the Espressif ESP32-DevKitC and the ESP-WROVER-KIT

This tutorial provides instructions for getting 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, or ESP32-WROVER-KIT. These versions of development boards are supported on FreeRTOS. For more information about these boards, see ESP32-DevKitC or ESP-WROVER-KIT on the Espressif website.

Note
Currently, the FreeRTOS port for ESP32-WROVER-KIT and ESP DevKitC does not support the following features:

- Symmetric multiprocessing (SMP).

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.

Prerequisites

Before you get started with FreeRTOS on your Espressif board, you need to set up your AWS account and permissions.

To create an AWS account, see Create and Activate an AWS Account.

To add an IAM user to your AWS account, see IAM User Guide. To grant your IAM user account access to AWS IoT and FreeRTOS, attach the following IAM policies to your IAM user account:

- `AmazonFreeRTOSFullAccess`
- `AWSIoTFullAccess`

To attach the AmazonFreeRTOSFullAccess policy to your IAM user

1. Browse to the IAM console, and from the navigation pane, choose **Users**.
2. Enter your user name in the search text box, and then choose it from the list.
3. Choose **Add permissions**.
4. Choose **Attach existing policies directly**.

5. In the search box, enter **`AmazonFreeRTOSFullAccess`**, choose it from the list, and then choose **Next: Review**.
6. Choose **Add permissions**.

To attach the AWSIoTFullAccess policy to your IAM user

1. Browse to the IAM console, and from the navigation pane, choose **Users**.
2. Enter your user name in the search text box, and then choose it from the list.
3. Choose **Add permissions**.
4. Choose **Attach existing policies directly**.
5. In the search box, enter **`AWSIoTFullAccess`**, choose it from the list, and then choose **Next: Review**.
6. Choose **Add permissions**.

For more information about IAM and user accounts, see IAM User Guide.

For more information about policies, see IAM Permissions and Policies.

Set up the Espressif hardware

See the ESP32-DevKitC Getting Started Guide for information about setting up the ESP32-DevKitC development board hardware.

See the ESP-WROVER-KIT Getting Started Guide for information about setting up the ESP-WROVER-KIT development board hardware.

> **Note**
> Do not proceed to the **Get Started** section of the Espressif guides. Instead, follow the steps below.

Set up your development environment

To communicate with your board, you need to download and install a toolchain.

Setting up the toolchain

To set up the toolchain, follow the instructions for your host machine's operating system:

- Standard Setup of Toolchain for Windows
- Standard Setup of Toolchain for macOS
- Standard Setup of Toolchain for Linux

> **Important**
> When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop and return to the instructions on this page.

Make sure that the `IDF_PATH` environment variable is cleared from your system before you continue. This environment variable is automatically set if you followed the "Get ESP-IDF" instructions under **Next Steps**.

> **Note**
> Version 3.3 of the ESP-IDF (the version that FreeRTOS uses) doesn't support the latest version of the ESP32 compiler. You must use the compiler that is compatible with version 3.3 of the ESP-IDF. See the previous links. To check the version of your compiler, run the following command.

```
xtensa-esp32-elf-gcc --version
```

Install CMake

The CMake build system is required to build the FreeRTOS demo and test applications for this device. FreeRTOS supports versions 3.13 and later.

You can download the latest version of CMake from CMake.org. Both source and binary distributions are available.

For more details about using CMake with FreeRTOS, see Using CMake with FreeRTOS (p. 80).

Establish a serial connection

To establish a serial connection between your host machine and the ESP32-DevKitC, you must install 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 some FTDI virtual COM port drivers. You can download these drivers from FTDI.

For more information, see Establish Serial Connection with ESP32. After you establish a serial connection, make a note of the serial port for your board's connection. You need it when you build the demo.

Download and configure FreeRTOS

After your environment is set up, you can download FreeRTOS from GitHub, or from the FreeRTOS console. See the README.md file for instructions.

Configure the FreeRTOS demo applications

1. If you are running macOS or Linux, open a terminal prompt. If you are running Windows, open **mingw32.exe**. (MinGW is a minimalist development environment for native Microsoft Windows applications.)
2. To verify that you have Python 2.7.10 or later installed, run **python --version**. The version installed is displayed. If you do not have Python 2.7.10 or later installed, you can install it from the Python website.
3. You need the AWS CLI to run AWS IoT commands. If you are running Windows, use the **easy_install awscli** to install the AWS CLI in the mingw32 environment.

   If you are running macOS or Linux, see Installing the AWS Command Line Interface.
4. Run **aws configure** and configure the AWS CLI with your AWS access key ID, secret access key, and default region name. 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 mingw32 environment, 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 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.

Valid security types are:

- `eWiFiSecurityOpen` (Open, no security)
- `eWiFiSecurityWEP` (WEP security)
- `eWiFiSecurityWPA` (WPA security)
- `eWiFiSecurityWPA2` (WPA2 security)

To run the configuration script

1. If you are running macOS or Linux, open a terminal prompt. If you are running Windows, open mingw32.exe.
2. Go 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.h` header file

  Note
  The certificate is hard-coded 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.

Build, flash, and run the FreeRTOS demo project

You can use CMake to generate the build files, Make to build the application binary, and Espressif's IDF utility to flash your board.

Build FreeRTOS on Linux and MacOS

(If you are using Windows, please see the next section.)

Use CMake to generate the build files, and then use Make to build the application.

To generate the demo application's build files with CMake

1. Change directories to the root of your FreeRTOS download directory.
2. Use the following command to generate the build files:

```
cmake -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 -S . -B your-build-directory
```

Note
If you want to build the application for debugging, add the `-DCMAKE_BUILD_TYPE=Debug` flag to this command.
If you want to generate the test application build files, add the `-DAFR_ENABLE_TESTS=1` flag.
The code provided by Espressif uses the lightweight IP (lwIP) stack as the default networking stack. To use the FreeRTOS+TCP networking stack instead, add the `-DAFR_ESP_FREERTOS_TCP` flag to the CMake command.
To add the lwIP dependency for non-vendor provided code, add the following lines to the CMake dependency file, `CMakeLists.txt`, for your custom WiFi component.

```
# Add a dependency on the bluetooth espressif component to the common component
set(COMPONENT_REQUIRES lwip)
```

To build the application with make

1. Change directories to the `build` directory.
2. Use the following command to build the application with Make:

```
make all -j4
```

Note
You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

Build FreeRTOS on Windows

On Windows, you must specify a build generator for CMake, otherwise CMake defaults to Visual Studio. Espressif officially recommends the Ninja build system because it works on Windows, Linux and MacOS. You must run CMake commands in a native Windows environment like cmd or PowerShell. Running CMake commands in a virtual Linux environment, like MSYS2 or WSL, is not supported.

Use CMake to generate the build files, and then use Make to build the application.

To generate the demo application's build files with CMake

1. Change directories to the root of your FreeRTOS download directory.
2. Use the following command to generate the build files:

```
cmake -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 -GNinja -S .
 -B build-directory
```

Note
If you want to build the application for debugging, add the `-DCMAKE_BUILD_TYPE=Debug` flag to this command.

If you want to generate the test application build files, add the `-DAFR_ENABLE_TESTS=1` flag.
The code provided by Espressif uses the lightweight IP (lwIP) stack as the default networking stack. To use the FreeRTOS+TCP networking stack instead, add the `-DAFR_ESP_FREERTOS_TCP` flag to the CMake command.
To add the lwIP dependency for non-vendor provided code, add the following lines to the CMake dependency file, `CMakeLists.txt`, for your custom WiFi component.

```
# Add a dependency on the bluetooth espressif component to the common component
set(COMPONENT_REQUIRES lwip)
```

To build the application

1. Change directories to the `build` directory.
2. Invoke Ninja to build the application:

```
ninja
```

Or, use the generic CMake interface to build the application:

```
cmake --build build-directory
```

Note
You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

Flash and run FreeRTOS

Use Espressif's IDF utility (*`freertos`*`/vendors/espressif/esp-idf/tools/idf.py`) to flash your board, run the application, and see logs.

To erase the board's flash, go to the *`freertos`* directory and use the following command:

```
./vendors/espressif/esp-idf/tools/idf.py erase_flash -B build-directory
```

To flash the application binary to your board, use `make`:

```
make flash
```

You can also use the IDF script to flash your board:

```
./vendors/espressif/esp-idf/tools/idf.py flash -B build-directory
```

To monitor:

```
./vendors/espressif/esp-idf/tools/idf.py monitor -p /dev/ttyUSB1 -B build-directory
```

Note
You can combine these commands. For example:

```
./vendors/espressif/esp-idf/tools/idf.py erase_flash flash monitor -p /dev/ttyUSB1
 -B build-directory
```

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

Run the Bluetooth Low Energy demos

FreeRTOS supports Bluetooth Low Energy connectivity.

To run the FreeRTOS demo project across Bluetooth Low Energy, you need to 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 the MQTT over Bluetooth Low Energy Demo Application.

For instructions about how to run the Wi-Fi provisioning demo on your board, see the Wi-Fi Provisioning Demo Application.

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 either from GitHub, or from the FreeRTOS console. If you're using git, 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 newer 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 -- amazon-freertos
# Commit the submodule change because it is pointing to a different revision now.
git add amazon-freertos
git commit -m "Update FreeRTOS to a new release"
```

Assuming 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
```

Here's 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)

add_executable(my_app src/main.c)

# Tell IDF build to link against this target.
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::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
```

To flash the application to your board, run

```
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 this:

```
====================Configuration for FreeRTOS====================
  Version:                 201910.00
  Git version:             201910.00-388-gcb3612cb7

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:        ble, ble_hal, ble_wifi_provisioning, common, crypto, defender,
```

```
                          dev_mode_key_provisioning, freertos_plus_tcp, greengrass,
                          https, kernel, mqtt, ota, pkcs11, pkcs11_implementation,
                          platform, secure_sockets, serializer, shadow, tls, wifi
  Enabled by user:        ble, ble_hal, ble_wifi_provisioning, defender, greengrass,
                          https, mqtt, ota, pkcs11, pkcs11_implementation, platform,
                          secure_sockets, shadow, wifi
  Enabled by dependency:  common, crypto, demo_base, dev_mode_key_provisioning,
                          freertos, freertos_plus_tcp, kernel, pkcs11_mbedtls,
                          secure_sockets_freertos_plus_tcp, serializer, tls, utils
  3rdparty dependencies:  http_parser, jsmn, mbedtls, pkcs11, tinycbor
  Available demos:        demo_ble, demo_ble_numeric_comparison, demo_defender,
                          demo_greengrass_connectivity, demo_https, demo_mqtt, demo_ota,
                          demo_shadow, demo_tcp, demo_wifi_provisioning
  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::mqtt`, `AFR::ota`, etc.

Add custom components to ESP-IDF

You can add more components to the ESP-IDF build environment. For example, assuming you want to add a component called `foo`, and your project looks like this:

```
- freertos
- components
  - foo
    - include
      - foo.h
    - src
      - foo.c
    - CMakeLists.txt
- src
  - main.c
- CMakeLists.txt
```

Here's an example of the CMakeLists.txt file for your component:

```
# include paths of this components.
set(COMPONENT_ADD_INCLUDEDIRS include)

# source files of this components.
set(COMPONENT_SRCDIRS src)
# Alternatively, use COMPONENT_SRCS to specify source files explicitly
# set(COMPONENT_SRCS src/foo.c)

# add this components, this will define a CMake library target.
register_component()
```

You can also specify dependencies using the standard CMake function `target_link_libraries`. Note that the target name for your component is stored in the variable `COMPONENT_TARGET`, defined by the ESP-IDF.

```
# add this component, this will define a CMake library target.
register_component()

# standard CMake function can be used to specify dependencies. ${COMPONENT_TARGET} is
 defined
# from esp-idf when you call register_component, by default it's
 idf_component_<folder_name>.
```

```
target_link_libraries(${COMPONENT_TARGET} PRIVATE AFR::mqtt)
```

For ESP components, this is done by setting 2 variables `COMPONENT_REQUIRES` and `COMPONENT_PRIV_REQUIRES`. See Build System (CMake) in the *ESP-IDF Programming Guide v3.3*.

```
# If the dependencies are from ESP-IDF, use these 2 variables. Note these need to be
# set before calling register_component().
set(COMPONENT_REQUIRES log)
set(COMPONENT_PRIV_REQUIRES lwip)
```

Then, in the top level `CMakeLists.txt` file, you tell ESP-IDF where to find these components. Insert the following lines anywhere before `add_subdirectory(freertos)`:

```
# 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/foo" ABSOLUTE
)
list(APPEND IDF_EXTRA_COMPONENT_DIRS ${EXTRA_COMPONENT_DIRS})
```

This component is now automatically linked to your application code by default. You should be able to 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/i` 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 will use the default file located at *freertos*`/vendors/espressif/boards/esp32/aws_demos/sdkconfig.defaults`.

Summary

If you have a project with a component called `foo`, 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)

add_executable(my_app 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/foo" 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::mqtt)
```

Troubleshooting

- If you are running macOS and the operating system does not recognize your ESP-WROVER-KIT, make sure you do not 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 crashes. For more information, see Automatically Decoding Addresses on the Espressif website.
- It is also possible to enable GDBstub for communication with gdb without requiring any special JTAG hardware. For more information, see Launch GDB for GDBStub on the Espressif website.
- For information about setting up an OpenOCD-based environment if JTAG hardware-based debugging is required, see the document *JTAG Debugging for ESP32* available on the Espressif website.
- If `pyserial` cannot 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. You can use the `-j4` switch on the **make** command to reduce the build time:

```
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 (p. 103) 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`.

For troubleshooting information, see Troubleshooting getting started (p. 79).

Debugging code on Espressif ESP32-DevKitC and ESP-WROVER-KIT

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

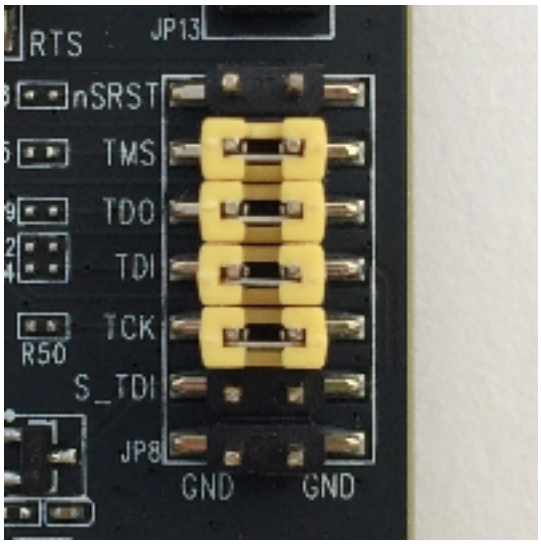
ESP-WROVER-KIT JTAG setup

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 C232HM MPSSE Cable Connection and Mechanical Details in the datasheet.

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

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 (p. 108). 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 choose **Properties**.

   Note
   The device might be listed as **USB Serial Port**.

   In the properties window, choose the **Details** tab to see the properties of the device. If the device is not 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 drop-down menu, choose **C232HM-DDHSL-0**.

d. In the target driver field, to the right of the green arrow, choose **WinUSB**.

e. From the drop-down box under the target driver field, choose the arrow, and then choose **Install Driver**. Choose **Replace Driver**.

7. Open a command prompt, navigate to `projects/espressif/esp32/make/aws_demos` and run:

   For ESP32-WROOM-32 and ESP32-WROVER:

   ```
   openocd.exe -f esp32_devkitj_v1.cfg -f esp-wroom-32.cfg
   ```

   For ESP32-SOLO-1:

   ```
   openocd.exe -f esp32_devkitj_v1.cfg -f esp-solo-1.cfg
   ```

   Leave this command prompt open.

8. Open a new command prompt, navigate to your `msys32` directory, and run `mingw32.exe`. In the mingw32 terminal, navigate to `projects/espressif/esp32/make/aws_demos` and run **make flash monitor**.

9. Open another mingw32 terminal, navigate to `projects/espressif/esp32/make/aws_demos` and wait until the demo starts running on your board. When it does, run `xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf`. The program should stop in the `main` function.

   Note
   The ESP32 supports a maximum of two break points.

Debugging on macOS

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. If you are 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
   ```

7. 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 this:

   ```
   DEVICE:
   ```

```
Product ID: product-ID
Vendor ID: vendor-ID (Future Technology Devices International Limited)
```

8. Open `projects/espressif/esp32/make/aws_demos/esp32_devkitj_v1.cfg`. 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.
9. Open a terminal window, navigate to `projects/espressif/esp32/make/aws_demos`, and use the following command to run OpenOCD.

   For ESP32-WROOM-32 and ESP32-WROVER:

   ```
   openocd -f esp32_devkitj_v1.cfg -f esp-wroom-32.cfg
   ```

   For ESP32-SOLO-1:

   ```
   openocd -f esp32_devkitj_v1.cfg -f esp-solo-1.cfg
   ```

10. Open a new terminal, and use the following command to load the FTDI serial port driver:

    ```
    sudo kextload -b com.FTDI.driver.FTDIUSBSerialDriver
    ```

11. Navigate to `projects/espressif/esp32/make/aws_demos`, and run the following command:

    ```
    make flash monitor
    ```

12. Open another new terminal, navigate to `projects/espressif/esp32/make/aws_demos`, and run the following command:

    ```
    xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf
    ```

    The program should stop at `main()`.

Debugging on Linux

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 this:

   ```
   $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 `projects/espressif/esp32/make/aws_demos`, and use the following command to run OpenOCD.

   For ESP32-WROOM-32 and ESP32-WROVER:

   ```
   openocd -f esp32_devkitj_v1.cfg -f esp-wroom-32.cfg
   ```

   For ESP32-SOLO-1:

   ```
   openocd -f esp32_devkitj_v1.cfg -f esp-solo-1.cfg
   ```

6. Open another terminal, navigate to `projects/espressif/esp32/make/aws_demos`, and run the following command:

   ```
   make flash monitor
   ```

7. Open another terminal, navigate to `projects/espressif/esp32/make/aws_demos`, and run the following command:

   ```
   xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf
   ```

   The program should stop in `main()`.

Getting started with the Espressif ESP32-WROOM-32SE (preview)

Follow this tutorial to get started with the Espressif ESP32-WROOM-32SE. Support for the ESP32-WROOM-32SE (with Microchip ATECC608A secure element) is in preview only and isn't part of the official FreeRTOS release. The ESP32-WROOM-32SE currently has limited availability for purchase. Contact sales@espressif.com to obtain a board.

Note
FreeRTOS port for ESP32-WROOM-32SE doesn't support the following features:

- Symmetric multiprocessing (SMP)
- Online Configuration Wizard (OCW)

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 need to set up your AWS account and permissions.

To create an AWS account, see Create and Activate an AWS Account.

To add an IAM user to your AWS account, see the Adding a User in the *IAM User Guide*. To grant your IAM user permission to AWS IoT and FreeRTOS, attach the following IAM managed policies to your IAM users:

- `AmazonFreeRTOSFullAccess`

  Allows full access to all of your IAM user's FreeRTOS resources (`freertos:*`).
- `AWSIoTFullAccess`

  Allows full access to all of your IAM user's IoT resources (`iot:*`).

To attach the `AmazonFreeRTOSFullAccess` policy to your IAM user

1. Navigate to the IAM console.
2. In the navigation pane, choose **Users**
3. Enter your user name in the search text box, and then choose it from the list.
4. Choose **Add permissions**.
5. Choose **Attach existing policies directly**.
6. In the search box, enter **`AmazonFreeRTOSFullAccess`**, choose it from the list, and then choose **Next: Review**.
7. Choose **Add permissions**.

To attach the AWSIoTFullAccess policy to your IAM user

1. Navigate to the IAM console.
2. In the navigation pane, choose **Users**.
3. Enter your user name in the search text box, and then choose it from the list.
4. Choose **Add permissions**.
5. Choose **Attach existing policies directly**.
6. In the search box, enter **`AWSIoTFullAccess`**, choose it from the list, and then choose **Next: Review**.
7. Choose **Add permissions**.

For more information about IAM, see the IAM User Guide.

For more information about policies, see IAM Permissions and Policies.

Set up the Espressif hardware

For information about setting up the ESP32-WROOM-32SE development board hardware, see the ESP32-DevKitC Getting Started Guide.

> **Note**
> Don't follow the **Get Started** section of the Espressif guides. Instead, follow the steps below.

Set up your development environment

To communicate with your board, you need to download and install a toolchain.

Set up the toolchain

To set up the toolchain, follow the instructions for your host machine's operating system:

- Standard Setup of Toolchain for Windows
- Standard Setup of Toolchain for macOS
- Standard Setup of Toolchain for Linux

> **Important**
> When you reach the "Get ESP-IDF" instructions under **Next Steps**, stop and return to the instructions on this page.

Make sure that the `IDF_PATH` environment variable is cleared from your system before you continue. This environment variable is automatically set if you followed the "Get ESP-IDF" instructions under **Next Steps**.

> **Note**
> Version 3.3 of the ESP-IDF (the version that FreeRTOS uses) doesn't support the latest version of the ESP32 compiler. You must use the compiler that is compatible with version 3.3 of the ESP-IDF. See the previous links. To check the version of your compiler, run the following command.

```
xtensa-esp32-elf-gcc --version
```

Install CMake

The CMake build system is required to build the FreeRTOS demo and test applications for this device. FreeRTOS supports versions 3.13 and later.

You can download the latest version of CMake from CMake.org. Source and binary distributions are available.

For more details about using CMake with FreeRTOS, see Using CMake with FreeRTOS (p. 80).

Establish a serial connection

1. 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.
2. Follow the steps to Establish a Serial Connection with ESP32.
3. After you establish a serial connection, make a note of the serial port for your board's connection. You need it when you build the demo.

Download and configure FreeRTOS

After you set up your environment, you can download FreeRTOS from GitHub. See the README.md file for instructions. The code for ESP32-WROOM32-SE is only available on GitHub on the development branch feature/esp32-wroom-32se.

> **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 (p. 75). Skip the last step **To format your AWS IoT credentials** and follow the steps below instead.

2. Microchip has provided several scripting tools to help with the setup of the ATECC608A parts. Navigate to the *freertos*/vendors/microchip/secure_elements/app/example_trust_chain_tool directory, and open the README.md file.

   Follow the instructions in the README.md file to provision your device. The steps include:

   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.
3. Load the CA certificate and device certificate onto the device by following the instructions for Developer-mode key provisioning (p. 77).

Build, flash, and run the FreeRTOS demo project

You can use CMake to generate the build files, Make to build the application binary, and Espressif's IDF utility to flash your board.

Build FreeRTOS on Linux or MacOS

If you're using Windows, you can skip to Build FreeRTOS on Windows (p. 116).

Use CMake to generate the build files, and then use Make to build the application.

To generate the demo application's build files with CMake

1. Navigate to the root of your FreeRTOS download directory.
2. In a command line window, enter the following command to generate the build files.

```
cmake -DVENDOR=espressif -DBOARD=esp32_plus_ecc608a_devkitc -DCOMPILER=xtensa-esp32 -
S . -B your-build-directory
```

Note
To build the application for debugging, add the -DCMAKE_BUILD_TYPE=Debug flag.
To generate the test application build files, add the -DAFR_ENABLE_TESTS=1 flag.
The code provided by Espressif uses the lightweight IP (lwIP) stack as the default networking stack. To use the FreeRTOS+TCP networking stack instead, add the -DAFR_ESP_FREERTOS_TCP flag to the CMake command.
To add the lwIP dependency for non-vendor provided code, add the following lines to the CMake dependency file, CMakeLists.txt, for your custom WiFi component.

```
# Add a dependency on the bluetooth espressif component to the common component
set(COMPONENT_REQUIRES lwip)
```

To build the application with Make

1. Navigate to the build directory.
2. In a command line window, enter the following command to build the application with Make.

```
make all -j4
```

Note
You must generate the build files with the **cmake** command every time you switch between the aws_demos project and the aws_tests project.

Build FreeRTOS on Windows

On Windows, you must specify a build generator for CMake. Otherwise, CMake defaults to Visual Studio. Espressif officially recommends the Ninja build system because it works on Windows, Linux, and MacOS. You must run CMake commands in a native Windows environment like cmd or PowerShell. Running CMake commands in a virtual Linux environment, such as MSYS2 or WSL, isn't supported.

Use CMake to generate the build files, and then use Make to build the application.

To generate the demo application's build files with CMake

1. Navigate to the root of your FreeRTOS download directory.
2. In a command line window, enter the following command to generate the build files.

```
cmake -DVENDOR=espressif -DBOARD=esp32_plus_ecc608a_devkitc -DCOMPILER=xtensa-esp32 -
GNinja -S . -B your-build-directory
```

Note
To build the application for debugging, add the `-DCMAKE_BUILD_TYPE=Debug` flag.
To generate the test application build files, add the `-DAFR_ENABLE_TESTS=1` flag.
The code provided by Espressif uses the lightweight IP (lwIP) stack as the default networking stack. To use the FreeRTOS+TCP networking stack instead, add the `-DAFR_ESP_FREERTOS_TCP` flag to the CMake command.
To add the lwIP dependency for non-vendor provided code, add the following lines to the CMake dependency file, `CMakeLists.txt`, for your custom WiFi component.

```
# Add a dependency on the bluetooth espressif component to the common component
set(COMPONENT_REQUIRES lwip)
```

To build the application

1. Navigate to the `build` directory.
2. In a command line window, enter the following command to invoke Ninja to build the application.

```
ninja
```

Or, use the generic CMake interface to build the application.

```
cmake --build your-build-directory
```

Note
You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

Flash and run FreeRTOS

Use Espressif's IDF utility (*freertos*`/vendors/espressif/esp-idf/tools/idf.py`) to flash your board, run the application, and see logs.

To erase the board's flash, navigate to the *freertos* directory and enter the following command.

```
./vendors/espressif/esp-idf/tools/idf.py erase_flash -B build-directory
```

To flash the application binary to your board, use `make`.

```
make flash
```

You can also use the IDF script to flash your board.

```
./vendors/espressif/esp-idf/tools/idf.py flash -B build-directory
```

To monitor:

```
./vendors/espressif/esp-idf/tools/idf.py monitor -p /dev/ttyUSB1 -B build-directory
```

Tip
You can also combine these commands.

```
./vendors/espressif/esp-idf/tools/idf.py erase_flash flash monitor -p /dev/ttyUSB1
 -B build-directory
```

Monitoring MQTT messages on the AWS Cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

Getting started with the Infineon XMC4800 IoT Connectivity Kit

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 (p. 72) 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.

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 (p. 80). (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 (p. 81).

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 (p. 118).

   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 (p. 118) that you established with the board.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 72) for instructions.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started (p. 79).

Getting started with the Infineon OPTIGA Trust X and XMC4800 IoT Connectivity Kit

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 (p. 117) 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 (p. 77). 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

You can use the MQTT client in the AWS IoT console to monitor the messages that your device sends to the AWS Cloud. You might want to set this up before the device runs the demo project.

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 77). 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 (p. 80). (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 (p. 81).
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 (p. 118).
   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 (p. 118) that you established with the board. Continue to the public key extraction step in Option #2: onboard private key generation (p. 77). 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 (p. 79).

Getting started with the Marvell MW320 AWS IoT Starter Kit

This tutorial provides instructions for getting started with the Marvell MW320 AWS IoT Starter Kit.

The Marvell MW320 AWS IoT Starter Kit includes the Marvell MW320, a Cortex M4 microcontroller development board that integrates 802.11b/g/n Wi-Fi on a single microcontroller chip. The kit is FCC-certified and available for sale. The MW320 module is also FCC-certified and available for customization and volume sale.

If you do not have the Marvell MW320 AWS IoT Starter Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

> **Note**
> In this tutorial, we use Ubuntu 16.04 for developing and debugging applications for the Marvell MW320. Other operating systems might work, but are not officially supported.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps (p. 72) 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.
5. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Set up your development environment

FreeRTOS includes some scripts for installing required third-party libraries, and for building and flashing applications to the board. These scripts are in the `vendors/marvell/WMSDK/mw320/sdk` directory.

The AWS IoT Starter Kit also includes pre-flashed wireless microcontroller demo project firmware.

In addition to the software that is bundled with the M320 AWS IoT Starter Kit and its FreeRTOS port, you must have the following software components installed on your host machine:

- The GCC toolchain, to cross compile your application. Versions `4_9_2015q3` and later are supported.
- OpenOCD, to access various JTAG functions for programming the board's flash memory, and for downloading and running firmware images in memory.
- The CMake build system. Versions 3.13 and later are supported.
- (Optional) A supported IDE, for application development and debugging.

Install required third-party libraries with installpkgs.sh

The `vendors/marvell/WMSDK/mw320/sdk/tools/bin/installpkgs.sh` script attempts to autodetect the machine type and install some required libraries, which include:

- C libraries
- A USB library
- An FTDI library
- ncurses
- Python
- LaTeX

> **Note**
> The `installpkgs.sh` script includes instructions for installing packages using `apt-get` for 32-bit and 64-bit Ubuntu environments and `yum` for 32-bit and 64-bit Fedora environments. If you have problems running the script on your distribution, open the script file, find the list of required packages, and install them manually.

With root privileges, issue the following command from the root directory of your FreeRTOS download:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/bin/installpkgs.sh
```

You can configure the permissions on your Linux host machine to allow `flashprog` and `ramload` operations without `sudo`. To do this, issue the following command:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/bin/perm_fix.sh
```

> **Note**
> If you are using the Eclipse IDE, you must configure these permissions.

Set up the toolchain

The FreeRTOS port for the this board is configured to use the GNU toolchain by default. For the Makefiles to invoke the correct compiler toolchain, the GNU compiler toolchain binaries must be included in the user's PATH variable. The GNU toolchain binaries must also be prefixed with `arm-none-eabi-`.

The GCC toolchain can be used with the GNU Debugger (GDB) for debugging with the OpenOCD software that interfaces with JTAG.

To set up the GCC toolchain on a Linux machine

1. Download the toolchain tarball from launchpad. The file name is `gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2`.
2. Copy the file to a directory of your choice. Make sure that there are no spaces in the directory path.
3. Untar the file with the following command:

   ```
   tar -vxf file-name
   ```

4. Add the toolchain binaries to your system PATH.

   For example, open the `.profile` file in your `$HOME` directory, and append the following line to the end of the file:

   ```
   PATH="$PATH:path/gcc-arm-none-eabit-4_9_2015_q3/bin"
   ```

Where *path* is the full directory path to the `gcc-arm-none-eabit-4_9_2015_q3` folder.

Note
Some distributions of Ubuntu include a Debian version of the GCC cross compiler. If your distribution includes a native cross compiler, remove it, and follow the steps to set up the GCC compiler toolchain.

Set up OpenOCD

OpenOCD version 0.9 is required. If an earlier version is installed on your host machine, remove it using your distribution's uninstall process.

You can install OpenOCD with standard Linux package managers. For example:

```
apt-get install openocd
```

The default version of OpenOCD installed on your machine depends on the version of your Linux kernel.

If you cannot use a package manager to install OpenOCD version 0.9, follow these steps:

1. Download the OpenOCD version 0.9 source code from OpenOCD.org.
2. Extract the `openocd` download, and then change directories to the extracted folder.
3. Enable FTDI and JLink:

   ```
   ./configure --enable-ftdi --enable-jlink
   ```

4. Make openocd:

   ```
   make install
   ```

Install CMake

The CMake build system is required to build the FreeRTOS demo and test applications for this device. FreeRTOS supports versions 3.13 and later.

You can download the latest version of CMake from CMake.org. Both source and binary distributions are available.

For more details about using CMake with FreeRTOS, see Using CMake with FreeRTOS (p. 80).

Establish a serial connection

To establish a serial connection between your host machine and your board

1. Attach one end of a USB cable to your host machine, and the other end to your board.

   Your host machine should detect the board. You can issue the **dmesg** command from the command line, or you can open the `/var/log/messages` file to verify that the board has been detected. A message similar to the following shows a successful connection:

   ```
   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
```

Note
Marvell development boards have an FTDI chip that exposes two USB interfaces to the host. The first interface (`ttyUSB0`) is to the MCU's JTAG functionality. The second interface (`ttyUSB1`) is to the MCU's physical UARTx port.

2. Open a serial connection to the `ttyUSB1` interface with the following settings:

Terminal Setting	Value
BAUD rate	115200
Data	8 bit
Parity	none
Stop	1 bit
Flow control	none

For example, if you are using minicom, do the following:

a. Start minicom in setup mode:

```
minicom -s
```

b. Go to **Serial Port Setup**.

c. Configure 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
```

Go to **Exit** to start showing messages from the serial console.

For more information about installing a terminal emulator to set up a serial connection, see Installing a terminal emulator (p. 79).

Build, flash, and run the FreeRTOS demo project

You can use CMake and the utility scripts included with the M320 port of FreeRTOS to build, flash, and run the FreeRTOS demo project from the command line. Or you can use an IDE to build your project.

Generate the demo build files with CMake

Issue the following command from the root of the FreeRTOS download to generate the demo build files with CMake:

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=0
```

or

```
cmake -DVENDOR=marvell -DBOARD=mw322 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=0
```

You should see output similar to the following:

```
==================Configuration for Amazon FreeRTOS==================
  Version:                1.4.7
  Git version:            v1.4.7-128-gc28d0a266

Target microcontroller:
  vendor:                 Marvell
  board:                  mw300_rd
  description:            Marvell Board for AmazonFreeRTOS
  family:                 Wireless Microcontroller
  data ram size:          512KB
  program memory size:    2MB

Host platform:
  OS:                     Darwin-17.7.0
  Toolchain:              arm-gcc
  Toolchain path:         /usr/local
  CMake generator:        Unix Makefiles

Amazon FreeRTOS modules:
  Modules to build:       bufferpool, crypto, freertos_plus_tcp, greengrass, kernel,
                          mqtt, pkcs11, secure_sockets, shadow, tls, wifi
  Enabled by user:        greengrass, mqtt, pkcs11, secure_sockets, shadow, wifi
  Enabled by dependency:  bufferpool, common, crypto, demo_base, freertos,
                          freertos_plus_tcp, kernel, pkcs11_mbedtls, tls, utils
  3rdparty dependencies:  jsmn, mbedtls, pkcs11
  Available demos:        demo_greengrass, demo_key_provisioning, demo_logging,
                          demo_mqtt_hello_world, demo_mqtt_pubsub, demo_shadow, demo_tcp
  Available tests:
======================================================================
```

Build the demo with make

Issue the following commands to build the demo:

```
cd build-directory
```

```
make all -j4
```

You should see output similar to the following:



You can use a similar set of commands to build a test project:

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=1
```

or

```
cmake -DVENDOR=marvell -DBOARD=mw322 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=1
```

```
cd build-directory
```

```
make all -j4
```

Note
You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

Flash the application

The `flashprog.py` script is used to program your board's flash memory. The script is written in Python 2.7.

Before you can flash the demo application image to the board, prepare the board's flash memory with a layout file and the Boot2 bootloader.

To load the layout file and boot2 bootloader

1. Change directories to the root of the FreeRTOS download.
2. Run the `flashprog.py` Python script with the `-l` and `--boot2` options:

   ```
   ./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` script writes a layout to the flash, according to the default layout configuration defined in `vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt`. The layout holds partitioned information about the flash.

The script also writes a bootloader to the flash. The bootloader is located at `vendors/marvell/WMSDK/mw320/sdk/boot2/bin/boot2.bin`. The bootloader loads the microcontroller's firmware image after it is flashed to the board.

You should see output similar to the following:

```
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
```

After you flash the layout file and bootloader to the board, flash some firmware to the board. The Wi-Fi chipset requires that its own firmware is present in flash memory.

To flash the Wi-Fi firmware

1. Change directories to the root of the FreeRTOS download.
2. Run the `flashprog.py` Python script with the `--wififw` option:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --wififw ./vendors/
marvell/WMSDK/mw320/wifi-firmware/mw30x/mw30x_uapsta_W14.88.36.p135.bin
```

The `flashprog` script flashes the firmware to the board.

You should see output similar to the following:

```
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
```

With the layout, bootloader, and Wi-Fi firmware flashed to the board, you can flash the demo application to the board and run it.

To flash and run the demo

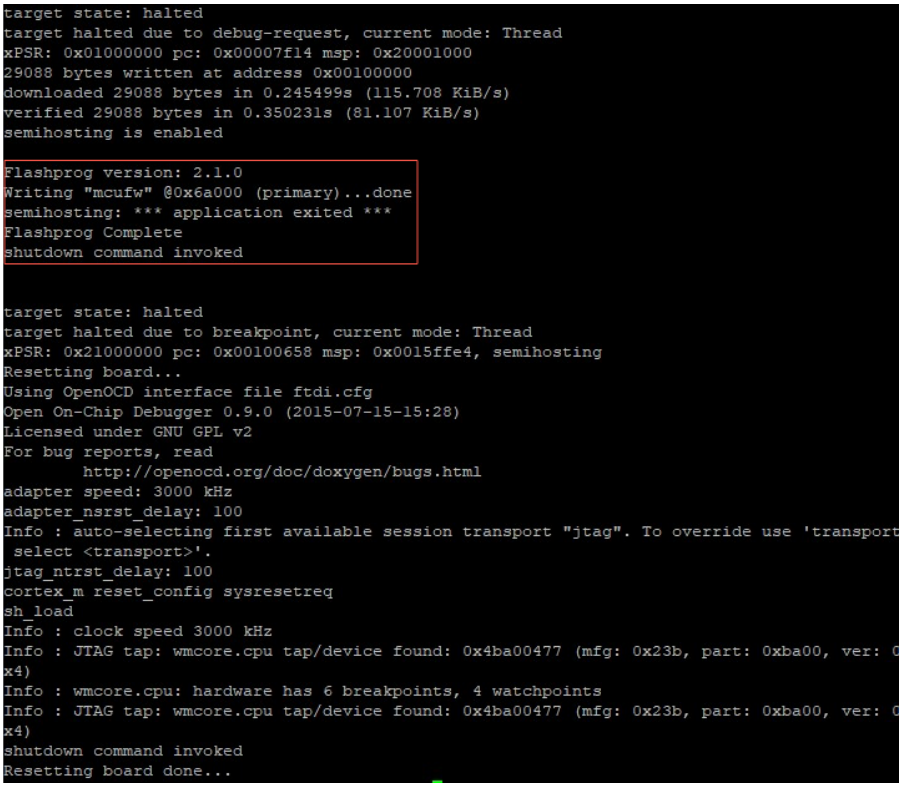
1. Change directories to the root of the FreeRTOS download.
2. Run the `flashprog.py` Python script with the `--mcufw` and `-r` options:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw ./build/marvell/
mw320/aws_demos.bin -r
```

The `flashprog` script flashes the demo to the board. Running the script with the `-r` option resets the board.

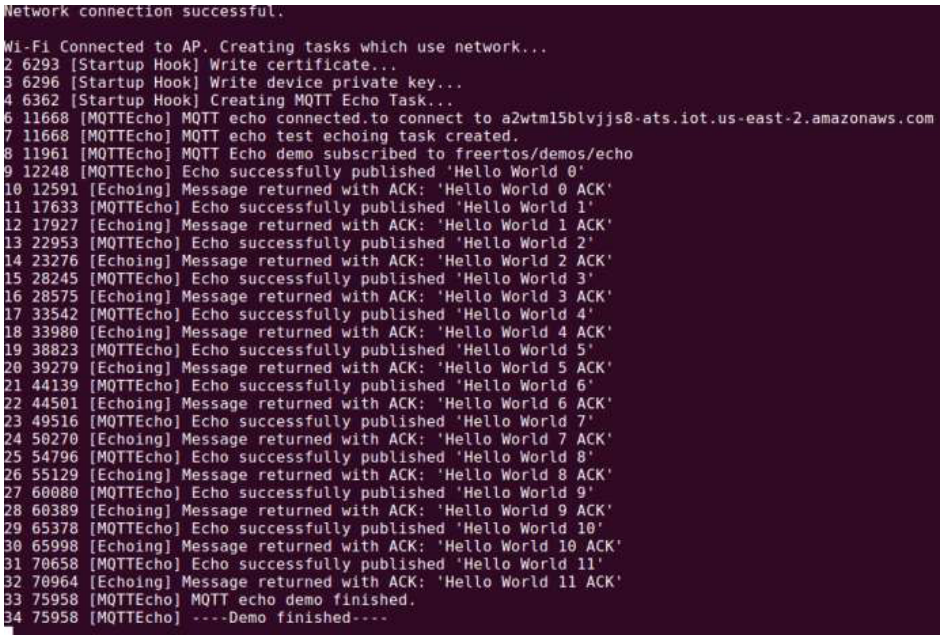
Reset the board.

You should see logs for the demo application. The output should be similar to the following:



Note
To flash the `aws_tests` application, use the same command, but specify the `aws_tests.bin` file instead of `aws_demos.bin`.
If you are only changing the application, you don't need to reload the layout, bootloader, and Wi-Fi firmware. If you change the layout, you might need to reload all of the components.

When you build, flash, and run the demo, you should see output similar to the following:



Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

Troubleshooting

Connecting to the GNU Debugger

To connect to the GNU Debugger (GDB)

1. Change directories:

```
cd freertos/vendors/marvell/WMSDK/mw320
```

2. Connect to GDB with the **arm-none-eabi-gdb** command:

```
arm-none-eabi-gdb -x ./sdk/tools/OpenOCD/gdbinit ../../../../build/vendors/marvell/
boards/mw300_rd/aws_demos.axf
```

   If you are debugging a FreeRTOS test application, target `aws_tests.axf` instead.

Loading the application to SRAM

You can load the demo to your device's static random-access memory (SRAM) and then execute the application on your device with the `ramload.py` script. Using `ramload.py` to load and execute the

application is a faster operation than loading to flash memory with the `flashprog.py` script, making it a more efficient approach to iterative development.

Note
The `ramload.py` script is written in Python 2.7.

To load to SRAM

1. Change directories to the root of the FreeRTOS download.
2. Run the `ramload.py` Python script on the `aws_demos.axf` file:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/ramload.py ./build/vendors/marvell/
boards/mw300_rd/aws_demos.axf
```

You should see logs for the demo application. The output should be similar to the following:

```
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
```

Note
Images loaded to SRAM are erased on reboot.

Enabling other logs

You might need to enable other logging messages to troubleshoot problems that you encounter while getting started with this board.

To enable board-specific logs

1. Open the `main.c` file of the project that you are working in (for example, `aws_tests` or `aws_demos`).
2. Enable the call to `wmstdio_init(UART0_ID, 0)` in the function `prvMiscInitialization`.

To enable Wi-Fi logs

1. Open `vendors/marvell/WMSDK/mw320/sdk/src/incl/autoconf.h`.
2. Enable the macro `CONFIG_WLCMGR_DEBUG`.

Using an IDE for development and debugging

Set up an IDE

You can use an IDE for developing and debugging applications, and for visualizing your projects.

If you are using the Eclipse IDE, for example, use the `perm_fix.sh` script to configure some permissions:

```
./vendors/marvell/WMSDK/mw320/tools/bin/perm_fix.sh
```

To set up Eclipse

1. Install Java Run Time Environment (JRE) from Oracle.

   JRE is required to run Eclipse. The JRE version (32-bit or 64-bit) must match the version of Eclipse (32-bit or 64-bit) that you install.
2. Download Eclipse IDE for C/C++ Developers from Eclipse.org. Eclipse versions 4.9.0 and later are supported.
3. Extract the downloaded archive folder, and then run the platform-specific Eclipse executable to start the IDE.

Build the demo with an IDE

You can open and build the demo project's build files in your IDE instead of building the demo directly from the command line with `make`. Opening the files in an IDE can help you visualize the project before you build it.

> **Note**
> You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

To build the project with Eclipse

1. Open Eclipse.
2. Choose your workspace to create a project.
3. On the **Select a wizard** page, expand **C/C++**, and choose **Makefile Project with Existing Code**.
4. On the **Import existing code** page, browse to the location of the `aws_demos` source code, choose `aws_demos`, and then choose **Finish**.
5. From the **Project Explorer**, right-click `aws_demos`, and then build the project.

   A successful build generates the `aws_demos.bin` executable.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started (p. 79).

Getting started with the Marvell MW322 AWS IoT Starter Kit

This tutorial provides instructions for getting started with the Marvell MW322 AWS IoT Starter Kit.

The Marvell MW322 AWS IoT Starter Kit includes the Marvell MW322, a Cortex M4 microcontroller development board that integrates 802.11b/g/n Wi-Fi on a single microcontroller chip. The kit is FCC-

certified and available for sale. The MW322 module is also FCC-certified and available for customization and volume sale.

If you do not have the Marvell MW322 AWS IoT Starter Kit, visit the AWS Partner Device Catalog to purchase one from our partner.

> **Note**
> In this tutorial, we use Ubuntu 16.04 for developing and debugging applications for the Marvell MW322. Other operating systems might work, but are not officially supported.

Before you begin, you must configure AWS IoT and your FreeRTOS download to connect your device to the AWS Cloud. See First steps (p. 72) 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.
5. Interacting with the application running on your board across a serial connection, for monitoring and debugging purposes.

Set up your development environment

FreeRTOS includes some scripts for installing required third-party libraries, and for building and flashing applications to the board. These scripts are in the `vendors/marvell/WMSDK/mw320/sdk` directory.

The AWS IoT Starter Kit also includes pre-flashed wireless microcontroller demo project firmware.

In addition to the software that is bundled with the M320 AWS IoT Starter Kit and its FreeRTOS port, you must have the following software components installed on your host machine:

- The GCC toolchain, to cross compile your application. Versions `4_9_2015q3` and later are supported.
- OpenOCD, to access various JTAG functions for programming the board's flash memory, and for downloading and running firmware images in memory.
- The CMake build system. Versions 3.13 and later are supported.
- (Optional) A supported IDE, for application development and debugging.

Install required third-party libraries with installpkgs.sh

The `vendors/marvell/WMSDK/mw320/sdk/tools/bin/installpkgs.sh` script attempts to autodetect the machine type and install some required libraries, which include:

- C libraries
- A USB library
- An FTDI library
- ncurses
- Python
- LaTeX

Note
The `installpkgs.sh` script includes instructions for installing packages using `apt-get` for 32-bit and 64-bit Ubuntu environments and `yum` for 32-bit and 64-bit Fedora environments. If you have problems running the script on your distribution, open the script file, find the list of required packages, and install them manually.

With root privileges, issue the following command from the root directory of your FreeRTOS download:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/bin/installpkgs.sh
```

You can configure the permissions on your Linux host machine to allow `flashprog` and `ramload` operations without `sudo`. To do this, issue the following command:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/bin/perm_fix.sh
```

Note
If you are using the Eclipse IDE, you must configure these permissions.

Set up the toolchain

The FreeRTOS port for the this board is configured to use the GNU toolchain by default. For the Makefiles to invoke the correct compiler toolchain, the GNU compiler toolchain binaries must be included in the user's PATH variable. The GNU toolchain binaries must also be prefixed with `arm-none-eabi-`.

The GCC toolchain can be used with the GNU Debugger (GDB) for debugging with the OpenOCD software that interfaces with JTAG.

To set up the GCC toolchain on a Linux machine

1. Download the toolchain tarball from launchpad. The file name is `gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2`.
2. Copy the file to a directory of your choice. Make sure that there are no spaces in the directory path.
3. Untar the file with the following command:

   ```
   tar -vxf file-name
   ```
4. Add the toolchain binaries to your system PATH.

   For example, open the `.profile` file in your `$HOME` directory, and append the following line to the end of the file:

   ```
   PATH="$PATH:path/gcc-arm-none-eabit-4_9_2015_q3/bin"
   ```

   Where *path* is the full directory path to the `gcc-arm-none-eabit-4_9_2015_q3` folder.

Note
Some distributions of Ubuntu include a Debian version of the GCC cross compiler. If your distribution includes a native cross compiler, remove it, and follow the steps to set up the GCC compiler toolchain.

Set up OpenOCD

OpenOCD version 0.9 is required. If an earlier version is installed on your host machine, remove it using your distribution's uninstall process.

You can install OpenOCD with standard Linux package managers. For example:

```
apt-get install openocd
```

The default version of OpenOCD installed on your machine depends on the version of your Linux kernel.

If you cannot use a package manager to install OpenOCD version 0.9, follow these steps:

1. Download the OpenOCD version 0.9 source code from OpenOCD.org.
2. Extract the `openocd` download, and then change directories to the extracted folder.
3. Enable FTDI and JLink:

   ```
   ./configure --enable-ftdi --enable-jlink
   ```

4. Make openocd:

   ```
   make install
   ```

Install CMake

The CMake build system is required to build the FreeRTOS demo and test applications for this device. FreeRTOS supports versions 3.13 and later.

You can download the latest version of CMake from CMake.org. Both source and binary distributions are available.

For more details about using CMake with FreeRTOS, see Using CMake with FreeRTOS (p. 80).

Establish a serial connection

To establish a serial connection between your host machine and your board

1. Attach one end of a USB cable to your host machine, and the other end to your board.

   Your host machine should detect the board. You can issue the **dmesg** command from the command line, or you can open the `/var/log/messages` file to verify that the board has been detected. A message similar to the following shows a successful connection:

   ```
   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
   ```

   Note
   Marvell development boards have an FTDI chip that exposes two USB interfaces to the host. The first interface (`ttyUSB0`) is to the MCU's JTAG functionality. The second interface (`ttyUSB1`) is to the MCU's physical UARTx port.

2. Open a serial connection to the `ttyUSB1` interface with the following settings:

Terminal Setting	Value
BAUD rate	115200
Data	8 bit
Parity	none
Stop	1 bit
Flow control	none

For example, if you are using minicom, do the following:

a. Start minicom in setup mode:

```
minicom -s
```

b. Go to **Serial Port Setup**.

c. Configure 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
```

Go to **Exit** to start showing messages from the serial console.

For more information about installing a terminal emulator to set up a serial connection, see Installing a terminal emulator (p. 79).

Build, flash, and run the FreeRTOS demo project

You can use CMake and the utility scripts included with the M320 port of FreeRTOS to build, flash, and run the FreeRTOS demo project from the command line. Or you can use an IDE to build your project.

Generate the demo build files with CMake

Issue the following command from the root of the FreeRTOS download to generate the demo build files with CMake:

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=0
```

or

```
cmake -DVENDOR=marvell -DBOARD=mw322 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=0
```

You should see output similar to the following:

```
====================Configuration for Amazon FreeRTOS====================
  Version:                1.4.7
  Git version:            v1.4.7-128-gc28d0a266

Target microcontroller:
  vendor:                 Marvell
  board:                  mw300_rd
  description:            Marvell Board for AmazonFreeRTOS
  family:                 Wireless Microcontroller
  data ram size:          512KB
  program memory size:    2MB

Host platform:
  OS:                     Darwin-17.7.0
  Toolchain:              arm-gcc
  Toolchain path:         /usr/local
  CMake generator:        Unix Makefiles

Amazon FreeRTOS modules:
  Modules to build:       bufferpool, crypto, freertos_plus_tcp, greengrass, kernel,
                          mqtt, pkcs11, secure_sockets, shadow, tls, wifi
  Enabled by user:        greengrass, mqtt, pkcs11, secure_sockets, shadow, wifi
  Enabled by dependency:  bufferpool, common, crypto, demo_base, freertos,
                          freertos_plus_tcp, kernel, pkcs11_mbedtls, tls, utils
  3rdparty dependencies:  jsmn, mbedtls, pkcs11
  Available demos:        demo_greengrass, demo_key_provisioning, demo_logging,
                          demo_mqtt_hello_world, demo_mqtt_pubsub, demo_shadow, demo_tcp
  Available tests:
========================================================================
```

Build the demo with make

Issue the following commands to build the demo:

```
cd build-directory
```

```
make all -j4
```

You should see output similar to the following:

```
[ 92%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/tinycbor/cborencoder.c.obj
[ 93%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/tinycbor/cborencoder_close
_container_checked.c.obj
[ 93%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/tinycbor/cborerrorstrings.
c.obj
[ 93%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/tinycbor/cborparser.c.obj
[ 94%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/tinycbor/cborparser_dup_st
ring.c.obj
[ 94%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/tinycbor/cborpretty.c.obj
[ 94%] Building C object CMakeFiles/afr_ota.dir/lib/third_party/jsmn/jsmn.c.obj
[ 95%] Building C object CMakeFiles/afr_ota.dir/demos/common/logging/aws_logging_task_dyna
mic_buffers.c.obj
[ 95%] Building C object CMakeFiles/afr_ota.dir/demos/common/demo_runner/aws_demo_runner.c
.obj
[ 95%] Linking C static library afr_ota.a
[ 95%] Built target afr_ota
[ 96%] Linking C executable aws_demos.axf
[100%] Built target aws_demos
marvell@pe-lt586:build$
```

You can use a similar set of commands to build a test project:

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=1
```

or

```
cmake -DVENDOR=marvell -DBOARD=mw322 -DCOMPILER=arm-gcc -S . -B build-directory -
DAFR_ENABLE_TESTS=1
```

```
cd build-directory
```

```
make all -j4
```

Note
You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

Flash the application

The `flashprog.py` script is used to program your board's flash memory. The script is written in Python 2.7.

Before you can flash the demo application image to the board, prepare the board's flash memory with a layout file and the Boot2 bootloader.

To load the layout file and `boot2` bootloader

1. Change directories to the root of the FreeRTOS download.
2. Run the `flashprog.py` Python script with the `-l` and `--boot2` options:

```
./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` script writes a layout to the flash, according to the default layout configuration defined in `vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt`. The layout holds partitioned information about the flash.

The script also writes a bootloader to the flash. The bootloader is located at `vendors/marvell/WMSDK/mw320/sdk/boot2/bin/boot2.bin`. The bootloader loads the microcontroller's firmware image after it is flashed to the board.

You should see output similar to the following:

```
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
```

After you flash the layout file and bootloader to the board, flash some firmware to the board. The Wi-Fi chipset requires that its own firmware is present in flash memory.

To flash the Wi-Fi firmware

1. Change directories to the root of the FreeRTOS download.
2. Run the `flashprog.py` Python script with the `--wififw` option:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --wififw ./vendors/
marvell/WMSDK/mw320/wifi-firmware/mw30x/mw30x_uapsta_W14.88.36.p135.bin
```

   The `flashprog` script flashes the firmware to the board.

   You should see output similar to the following:

```
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
```

With the layout, bootloader, and Wi-Fi firmware flashed to the board, you can flash the demo application to the board and run it.

To flash and run the demo

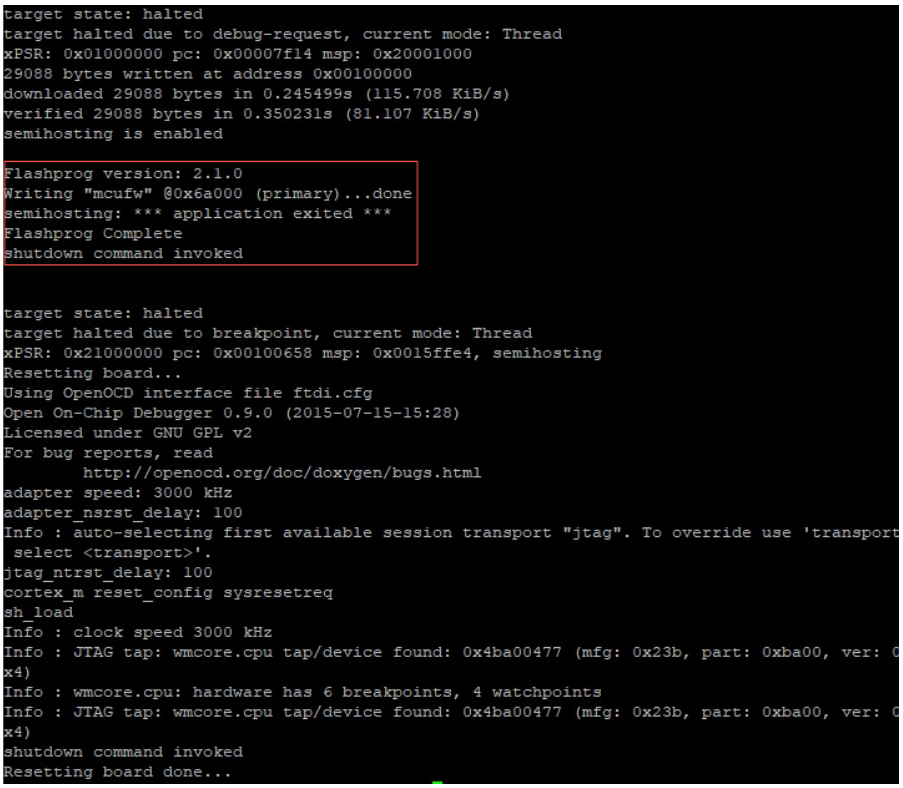
1. Change directories to the root of the FreeRTOS download.
2. Run the `flashprog.py` Python script with the `--mcufw` and `-r` options:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw ./build//marvell/
mw320/aws_demos.bin -r
```

   The `flashprog` script flashes the demo to the board. Running the script with the `-r` option resets the board.

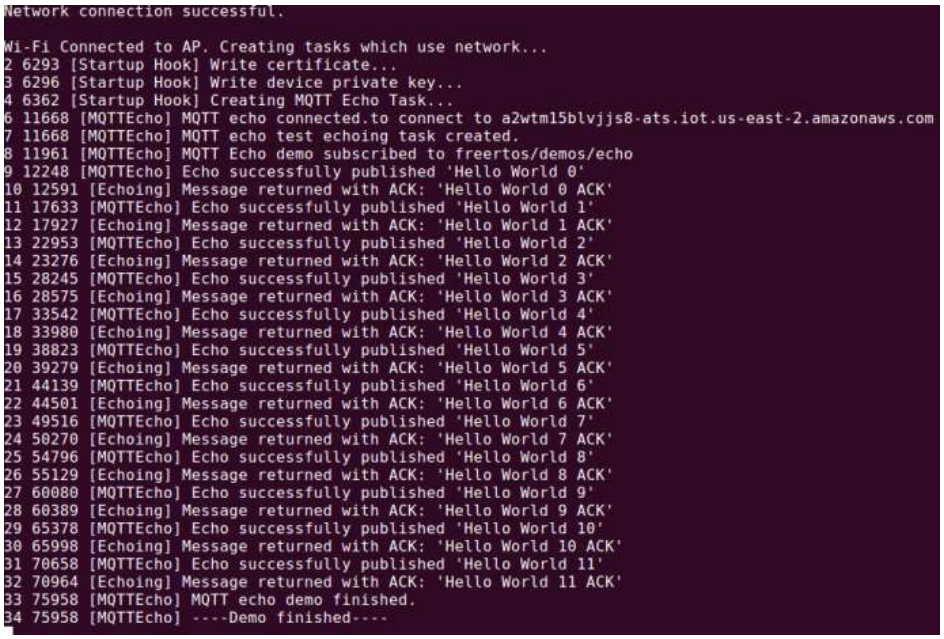
   Reset the board.

   You should see logs for the demo application. The output should be similar to the following:



Note
To flash the `aws_tests` application, use the same command, but specify the `aws_tests.bin` file instead of `aws_demos.bin`.
If you are only changing the application, you don't need to reload the layout, bootloader, and Wi-Fi firmware. If you change the layout, you might need to reload all of the components.

When you build, flash, and run the demo, you should see output similar to the following:



Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

Troubleshooting

Connecting to the GNU Debugger

To connect to the GNU Debugger (GDB)

1. Change directories:

```
cd freertos/vendors/marvell/WMSDK/mw320
```

2. Connect to GDB with the **arm-none-eabi-gdb** command:

```
arm-none-eabi-gdb -x ./sdk/tools/OpenOCD/gdbinit ../../../../build/vendors/marvell/
boards/mw300_rd/aws_demos.axf
```

   If you are debugging a FreeRTOS test application, target `aws_tests.axf` instead.

Loading the application to SRAM

You can load the demo to your device's static random-access memory (SRAM) and then execute the application on your device with the `ramload.py` script. Using `ramload.py` to load and execute the

application is a faster operation than loading to flash memory with the `flashprog.py` script, making it a more efficient approach to iterative development.

Note
The `ramload.py` script is written in Python 2.7.

To load to SRAM

1. Change directories to the root of the FreeRTOS download.
2. Run the `ramload.py` Python script on the `aws_demos.axf` file:

```
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/ramload.py ./build/vendors/marvell/
boards/mw300_rd/aws_demos.axf
```

You should see logs for the demo application. The output should be similar to the following:

```
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
```

Note
Images loaded to SRAM are erased on reboot.

Enabling other logs

You might need to enable other logging messages to troubleshoot problems that you encounter while getting started with this board.

To enable board-specific logs

1. Open the `main.c` file of the project that you are working in (for example, `aws_tests` or `aws_demos`).
2. Enable the call to `wmstdio_init(UART0_ID, 0)` in the function `prvMiscInitialization`.

To enable Wi-Fi logs

1. Open `vendors/marvell/WMSDK/mw320/sdk/src/incl/autoconf.h`.
2. Enable the macro `CONFIG_WLCMGR_DEBUG`.

Using an IDE for development and debugging

Set up an IDE

You can use an IDE for developing and debugging applications, and for visualizing your projects.

If you are using the Eclipse IDE, for example, use the `perm_fix.sh` script to configure some permissions:

```
./vendors/marvell/WMSDK/mw320/tools/bin/perm_fix.sh
```

To set up Eclipse

1. Install Java Run Time Environment (JRE) from Oracle.

   JRE is required to run Eclipse. The JRE version (32-bit or 64-bit) must match the version of Eclipse (32-bit or 64-bit) that you install.
2. Download Eclipse IDE for C/C++ Developers from Eclipse.org. Eclipse versions 4.9.0 and later are supported.
3. Extract the downloaded archive folder, and then run the platform-specific Eclipse executable to start the IDE.

Build the demo with an IDE

You can open and build the demo project's build files in your IDE instead of building the demo directly from the command line with `make`. Opening the files in an IDE can help you visualize the project before you build it.

> **Note**
> You must generate the build files with the **cmake** command every time you switch between the `aws_demos` project and the `aws_tests` project.

To build the project with Eclipse

1. Open Eclipse.
2. Choose your workspace to create a project.
3. On the **Select a wizard** page, expand **C/C++**, and choose **Makefile Project with Existing Code**.
4. On the **Import existing code** page, browse to the location of the `aws_demos` source code, choose `aws_demos`, and then choose **Finish**.
5. From the **Project Explorer**, right-click `aws_demos`, and then build the project.

   A successful build generates the `aws_demos.bin` executable.

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started (p. 79).

Getting started with the MediaTek MT7697Hx development kit

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 (p. 72) 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

To establish a serial connection with the MediaTek MT7697Hx Development Kit, you must install the Arm Mbed Windows serial port driver. You can download the driver from Mbed. Follow the steps on the **Windows serial driver** page to download and install the driver for the MediaTek MT7697Hx Development Kit.

After you install the driver, 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.

> **Note**
> If you are having trouble connecting to your board after you install the driver, you might need to reboot your machine.

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.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 ) ; }
```

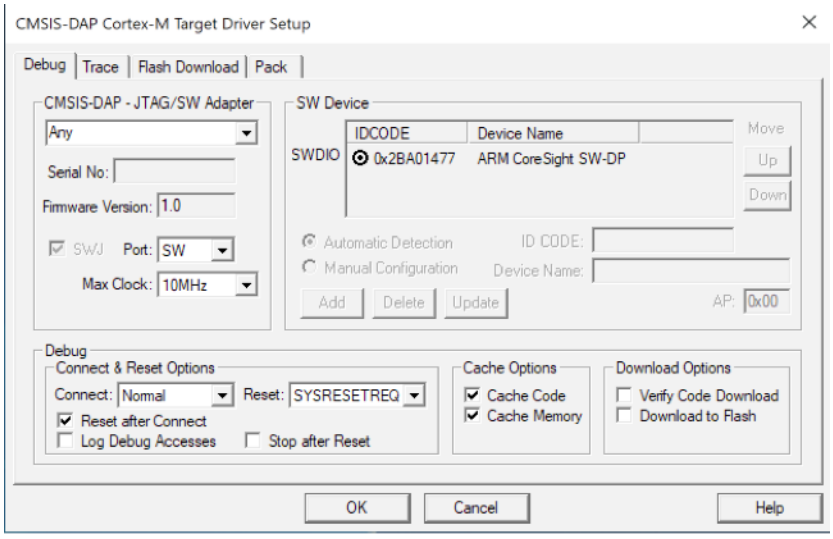
11. In the **Call Stack + Locals** window, change the value for `wait_ice` to `0`.
12. Set breakpoints in your project's source code, and run the code.

Troubleshooting the IDE debugger settings

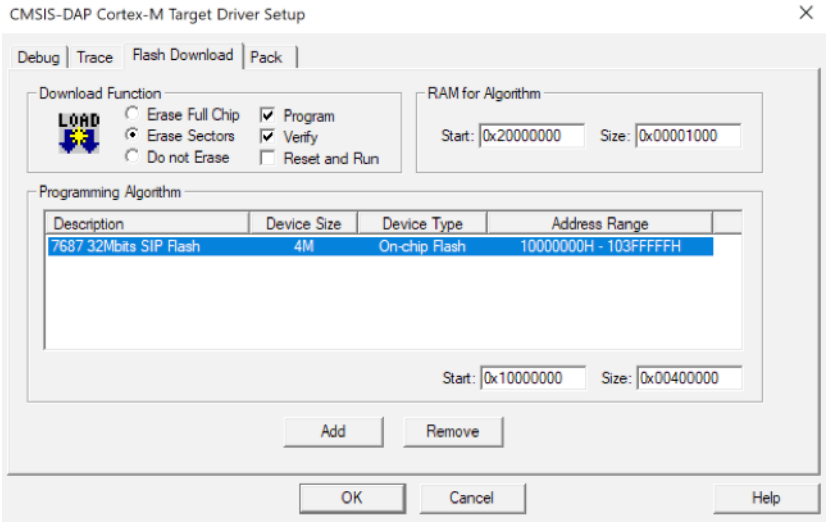
If you are having trouble debugging an application, your debugger settings might be incorrect.

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 (p. 79).

Getting started with the Microchip Curiosity PIC32MZ EF

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 (p. 72) 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.

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 (p. 75).
   - 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.
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 (p. 75).
   - 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.
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:
   - 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

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.

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 folder to contain the generated build files (*`build-folder`*).
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. (For example: `C:\Program Files (x86)\Microchip\MPLABX\v5.35\mplab_platform\bin` and `C:\Program Files\Microchip\xc32\v2.40\bin` respectively.)
3. Change directories to the build directory (*`build-folder`*), and run `make` from that directory.

For more information, see Using CMake with FreeRTOS (p. 80).

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

Troubleshooting

For troubleshooting information, see Troubleshooting getting started (p. 79).

Getting started with the Nordic nRF52840-DK

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 (p. 237).

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 the FreeRTOS Bluetooth Low Energy Mobile SDK Demo application

1. Follow the instructions in Mobile SDKs for FreeRTOS Bluetooth devices (p. 208) 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 (p. 239) 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 (p. 79).
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 (p. 79).

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.

Configure your project

To run 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 (p. 237).

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 **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_demos_config.h, and define CONFIG_MQTT_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.
   - NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE

     Attribute Table size in bytes. The size must be a multiple of 4.

   (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. 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 (p. 79).

Getting started with the Nuvoton NuMaker-IoT-M487

This tutorial provides instructions for getting started with the Nuvoton NuMaker-IoT-M487 development board. The Nuvoton NuMaker-IoT-M487 development board is embedded with the NuMicro M487 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 (p. 72). 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_V3.00.6951** (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 (p. 161) 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** to open the MQTT client.
3. In **Subscription topic**, enter `iotdemo/#`, 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 (p. 80), 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** to open the MQTT client.
3. In **Subscription topic**, enter `iotdemo/#`, 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.

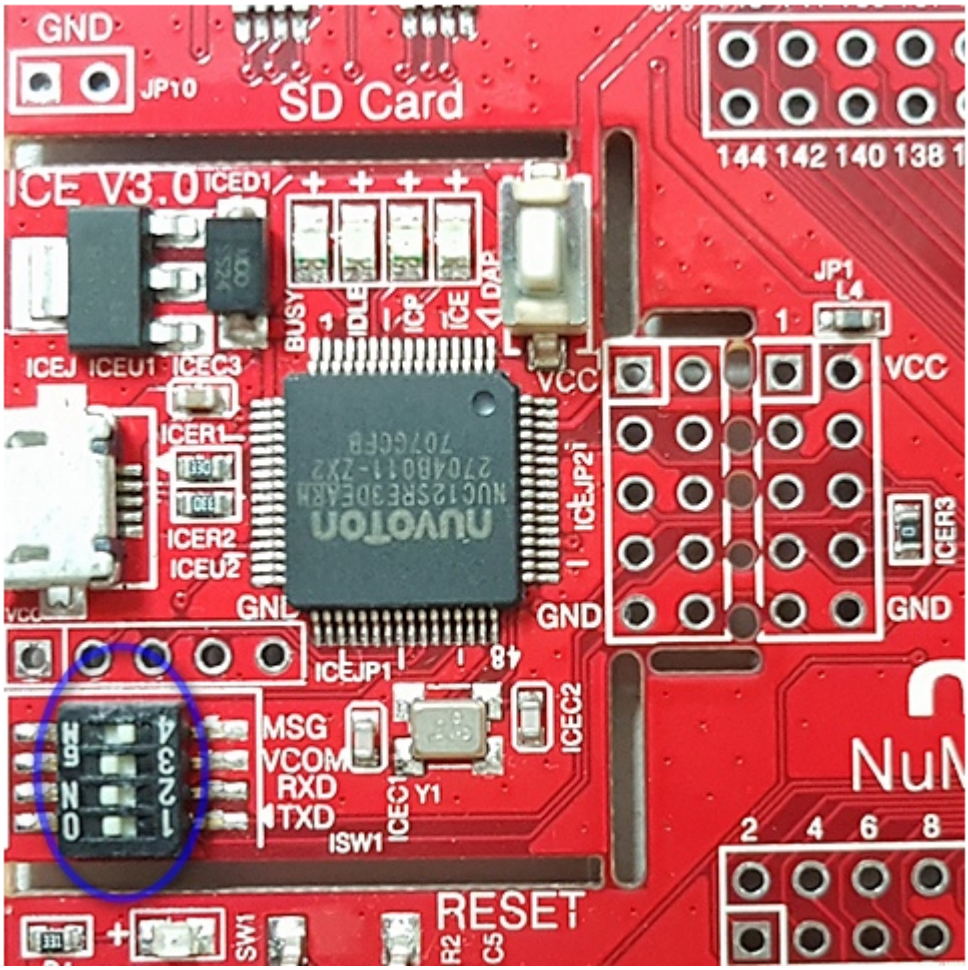
   ```
   cmake --build BUILD_FOLDER
   ```

   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 (p. 161).)
6. Open a terminal emulator to view the messages over UART. Follow the instructions at Installing a terminal emulator (p. 79).
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.

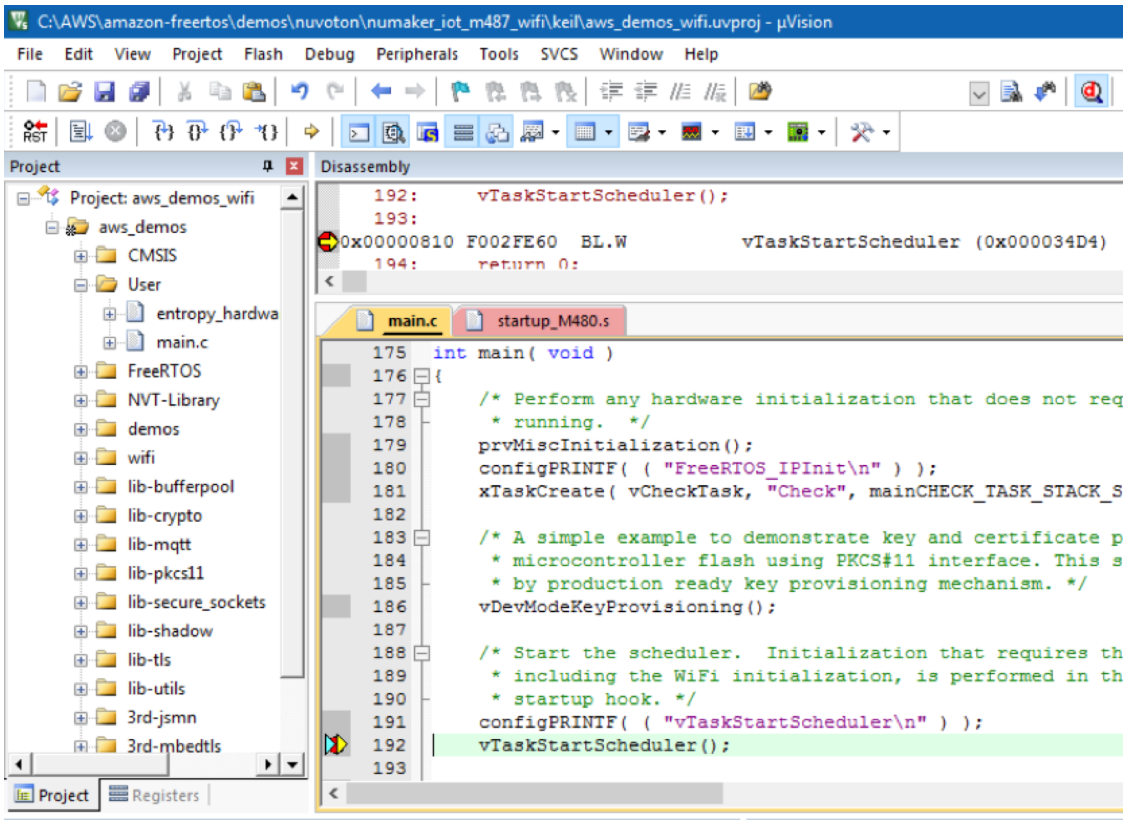
Debugging FreeRTOS projects in Keil µVision

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 (p. 159).
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.

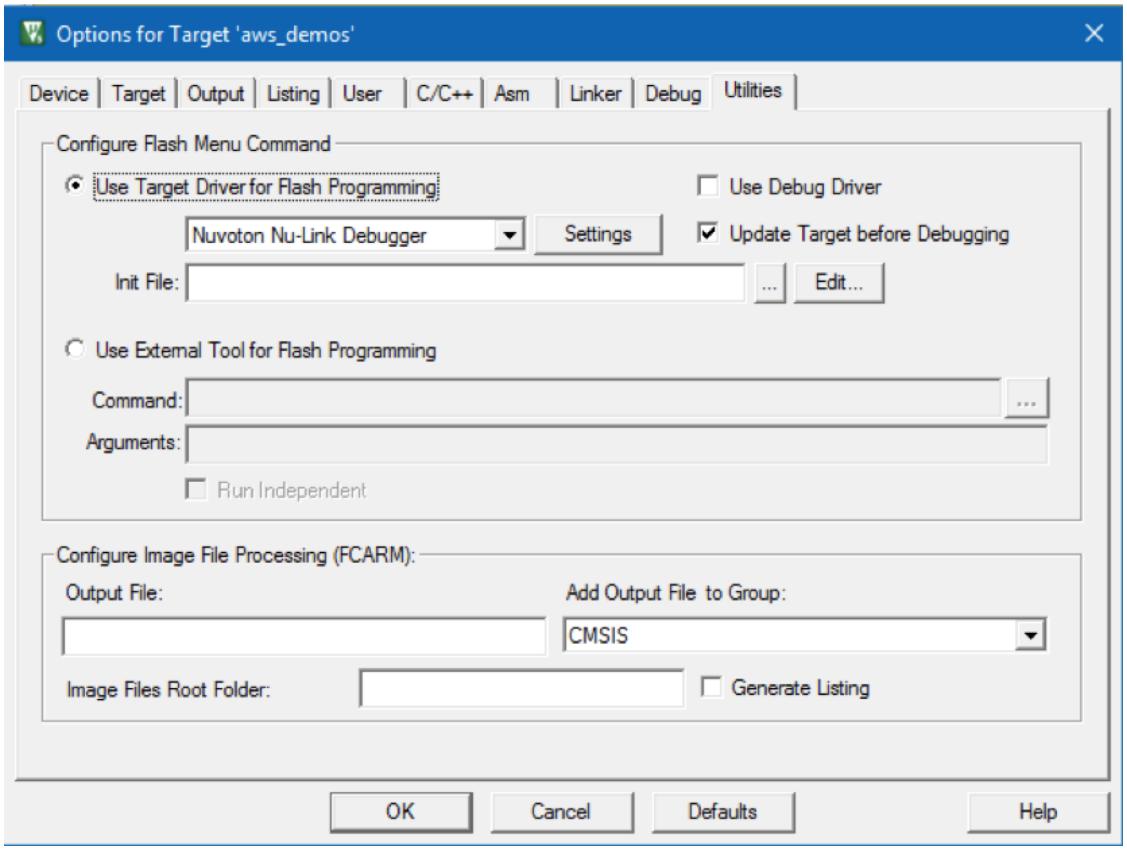


Troubleshooting µVision debug settings

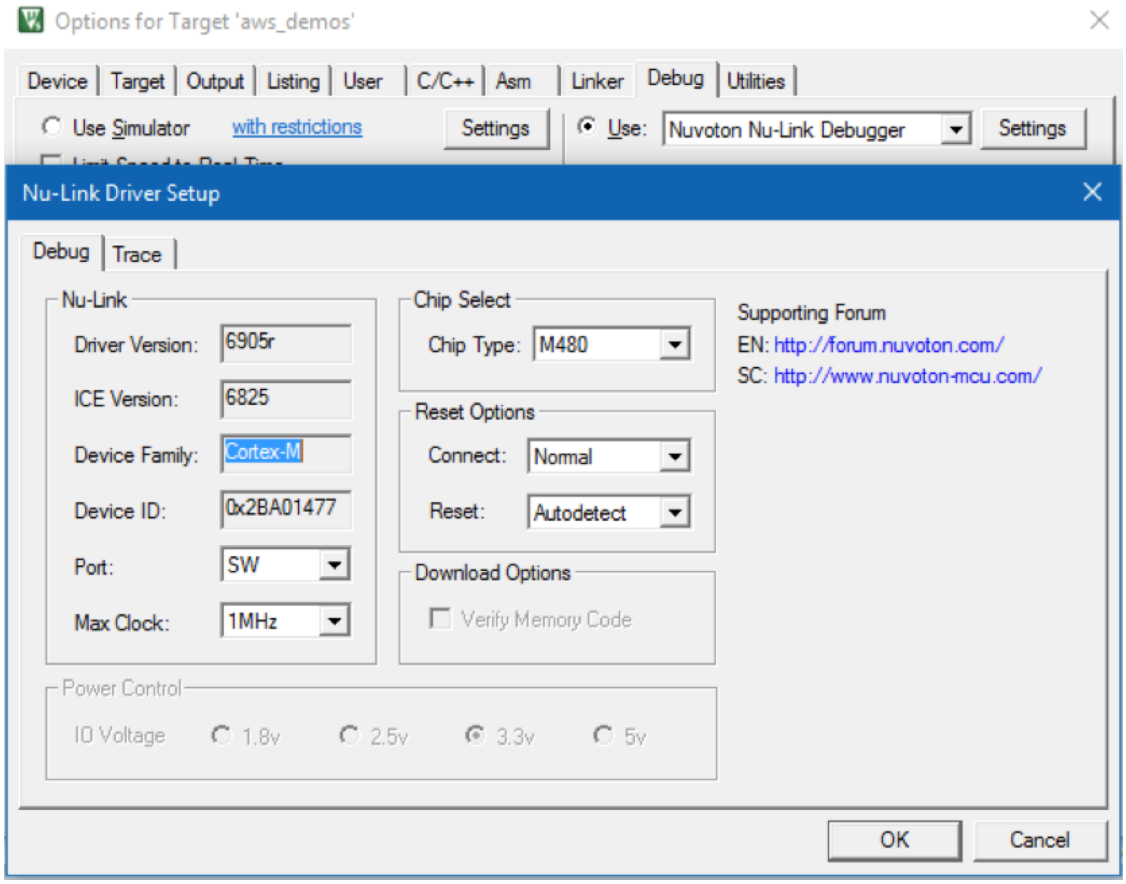
If you encounter problems while debugging an application, check that your debug settings are set correctly in Keil µVision.

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

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 (p. 72) 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 Software for NXP Kits and choose **Download Software**.

   Note
   IAR Embedded Workbench for ARM requires Microsoft Windows.
2. Unzip and run the installer. 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.

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**.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

Troubleshooting

For general troubleshooting information about Getting Started with FreeRTOS, see Troubleshooting getting started (p. 79).

Getting started with the Renesas Starter Kit+ for RX65N-2MB

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 (p. 72) 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

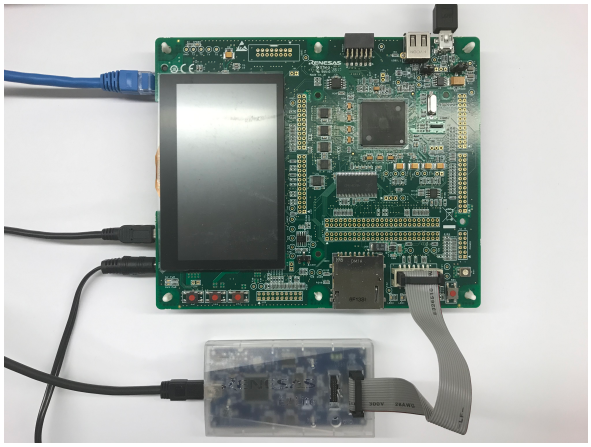
1. Connect the positive +5V power adapter to the PWR connector on the RSK+ for RX65N-2MB.
2. Connect your computer to the USB2.0 FS port on the RSK+ for RX65N-2MB.
3. Connect your computer to the USB-to-serial port on the RSK+ for RX65N-2MB.
4. Connect a router or internet-connected Ethernet port to 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 user name and password 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.

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.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 79).

Getting started with the STMicroelectronics STM32L4 Discovery Kit IoT Node

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 and your FreeRTOS download to connect your device to the AWS Cloud. See First steps (p. 72) 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.

Run the FreeRTOS demo project

1. Use a USB cable to connect your STMicroelectronics STM32L4 Discovery Kit IoT Node to your computer.
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 (p. 80).

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 79).

Getting started with the Texas Instruments CC3220SF-LAUNCHXL

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 (p. 72) 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 (p. 75).
- 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 UniFlash CC3x20, CC3x35 SimpleLink Wi-Fi and Internet of Things Programmer'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**.

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 UniFlash CC3x20, CC3x35 SimpleLink Wi-Fi and Internet of Things Programmer'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 (p. 80).

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 79).

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 (p. 72) 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 WinPCap.
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.

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), 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 will need to select a network interface (p. 177).

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.) Leave 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 as 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 will need to select a network interface (p. 177).

For more information about using CMake with FreeRTOS, see Using CMake with FreeRTOS (p. 80).

Configure your network interface

On the first run of the demo project, you must select the network interface to use. The program enumerates your network interfaces. Find the number for your hard-wired Ethernet interface. The output should look like this:

```
0 0 [None] FreeRTOS_IPInit
1 0 [None] vTaskStartScheduler
1. rpcap://\Device\NPF_{AD01B877-A0C1-4F33-8256-EE1F4480B70D}
(Network adapter 'Intel(R) Ethernet Connection (4) I219-LM' on local host)
```

```
2. rpcap://\Device\NPF_{337F7AF9-2520-4667-8EFF-2B575A98B580}
(Network adapter 'Microsoft' on local host)

The interface that will be opened is set by "configNETWORK_INTERFACE_TO_USE", which
should be defined in FreeRTOSConfig.h

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 example above, 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. For more information, see the *WinPcap FAQ* entry Q-16: Which network adapters are supported by WinPcap? .

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 79).

Getting started with the Xilinx Avnet MicroZed Industrial IoT Kit

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 (p. 72) for instructions. In this tutorial, the path to the FreeRTOS download directory is referred to as *`freertos`*.

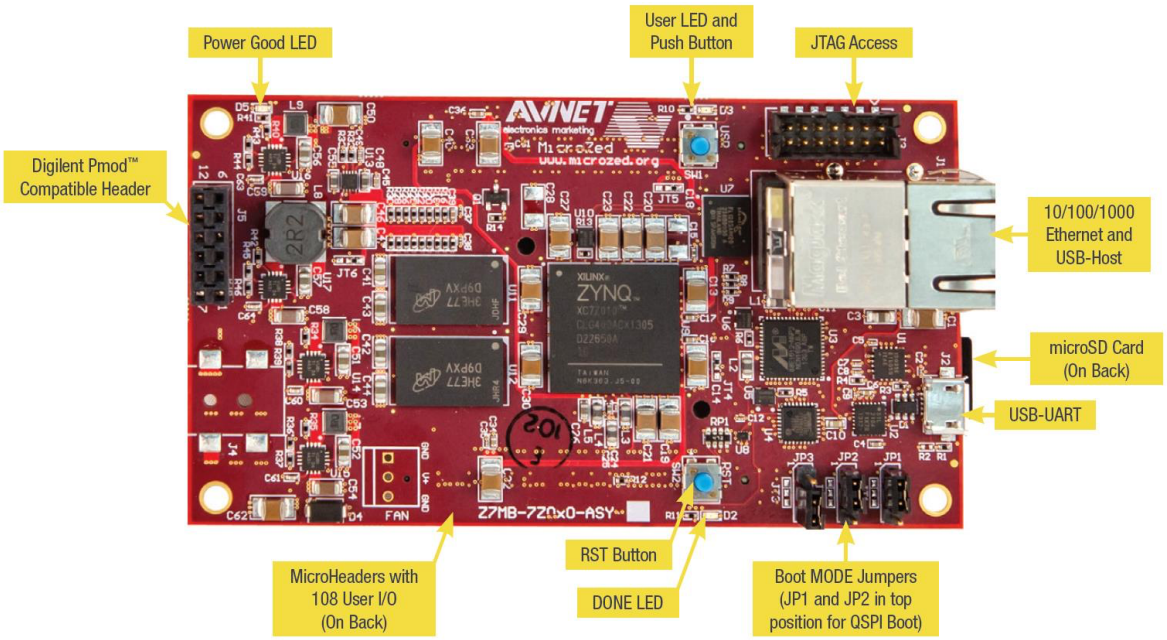
Overview

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 user name and password 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.

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.

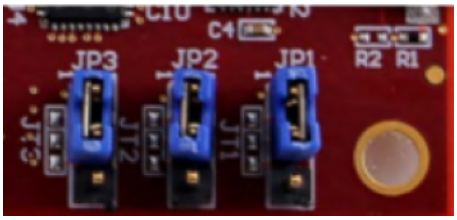
   Note
   To build all projects without cleaning them, choose **Project**, and then choose **Build All**.
   To build individual projects, select the project you want to build, choose **Project**, and then choose **Build Project**.

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

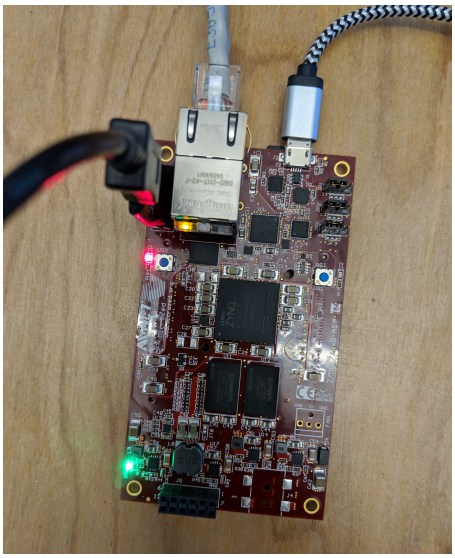
1. Set your MicroZed board's boot mode jumpers to the JTAG boot mode.



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:



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 (p. 181) and Run the FreeRTOS demo project (p. 182).

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 (p. 179). 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.

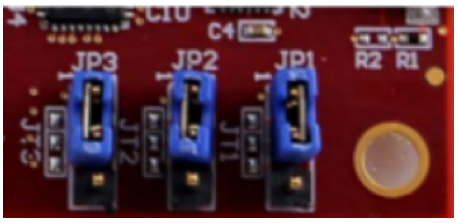
1. Copy the `BOOT.bin` file to the MicroSD card.
2. Insert the card into the MicroSD card slot directly under the USB-UART port.
3. Set the MicroZed boot mode jumpers to SD boot mode.



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.

Monitoring MQTT messages on the cloud

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.

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 (p. 181).

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 (p. 79).

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.

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	Description	Reference
Bluetooth Low Energy	Using the FreeRTOS Bluetooth Low Energy API, your microcontroller can communicate with the AWS IoT MQTT broker through a gateway device. For more information, see Bluetooth Low Energy library (p. 199).	
Over-the-air	The FreeRTOS	

Library	Description	Reference
AWS IoT Over-the-Air Update	The AWS IoT Over-the-Air (OTA) Agent library connects your FreeRTOS device to the AWS IoT OTA Agent. For more information, see OTA Agent library (p. 222).	API Reference
FreeRTOS +POSIX	You can use the FreeRTOS +POSIX library to port POSIX-compliant applications to the FreeRTOS ecosystem. For more information, see FreeRTOS +POSIX.	API Reference

Library	Description	Reference
Secure Sockets	Secure Sockets is a library that provides a simple, consistent, and portable API for network communication.	For more information, see Secure Sockets library (p. 226).
FreeRTOS +TCP	FreeRTOS +TCP is a scalable, open source and thread safe TCP/IP stack for FreeRTOS.	For more information, see FreeRTOS +TCP.

Library	Description	Reference
Wi-Fi	The FreeRTOS Wi-Fi library enables you to interface with your microcontroller's lower-level wireless stack. For more information, see Wi-Fi library (p. 230).	API Reference

Library	Description	Reference
PKCS #11	FreeRTOS PKCS #11 library is a reference implementation of the Public Key Cryptography Standard #11, to support provisioning and TLS client authentication. For more information, see Public Key Cryptography Standard (PKCS) #11 library (p. 224).	
TLS	For more information, see Transport Layer Security (p. 230).	
Common I/O	For more information, see I/O (p. 234).	API Reference

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 the AWS IoT Device SDK for Embedded C, see AWS IoT Device SDK for Embedded C (p. 69).

FreeRTOS application libraries

Library	Description	API Reference
AWS IoT Device Defender library	AWS IoT Device Defender library connects your FreeRTOS device to AWS IoT Device Defender. For more information, see AWS IoT Device Defender library (p. 210).	AWS IoT Device Defender library
AWS IoT Greengrass library	AWS IoT Greengrass library connects your FreeRTOS device to AWS IoT Greengrass.	AWS IoT Greengrass

Library	Description	Reference
		For more information, see AWS IoT Greengrass Discovery library (p. 214).

Library	Description	API Reference
MQTT (v3.1.0) library	The FreeRTOS MQTT 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 FreeRTOS MQTT library version 1.0.0, see MQTT library, version 1.0.0 (p. 218). For more information about the FreeRTOS MQTT	MQTT (v3.1.0) library API Reference MQTT (v2.0.0) C SDK API Reference

Library	Description	Reference
	library version 2.0.0, see MQTT library, version 2.0.0 (p. 216).	
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 (p. 212).	

FreeRTOS common libraries

The following common libraries extend the kernel functionality with additional data structures and functions for embedded application development. These libraries are often dependencies of the FreeRTOS porting and application libraries.

Note
The AWS IoT Device SDK for Embedded C includes common libraries with APIs and functionality identical to these libraries. For an API reference, see the AWS IoT Device SDK C API Reference.

FreeRTOS common libraries

Library	Description	Reference
Atomic Operations		

Library	Description
Atomic	For more information, see the Atomic API Reference.
Linear Containers	For more information, see the Linear Containers API Reference.
Logging	For more information, see the Logging API Reference.
Static Memory	For more information, see the Static Memory API Reference.
Task Pool	For more information, see the Task Pool API Reference.

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.

Common libraries

FreeRTOS includes some common libraries that extend the kernel functionality with additional data structures and functions for embedded application development. These libraries are often dependencies of other FreeRTOS libraries.

Topics

- Atomic operations (p. 195)
- Linear Containers library (p. 195)
- Logging library (p. 196)
- Static Memory library (p. 196)
- Task Pool library (p. 196)

Atomic operations

Overview

Atomic operations ensure non-blocking synchronization in concurrent programming. You can use atomic operations to solve performance issues that are caused by asynchronous operations that act on shared memory. FreeRTOS supports atomic operations, as implemented in the `iot_atomic.h` header file.

The `iot_atomic.h` header file includes two implementations for atomic operations:

- Disabling interrupt globally.

  This implementation is available to all FreeRTOS platforms.
- ISA native atomic support.

  This implementation is only available to platforms that compile with GCC, version 4.7.0 and higher, and have ISA atomic support. For information about GCC built-in functions, see Built-in Functions for Memory Model Aware Atomic Operations.

Initialization

Before you use FreeRTOS Atomic Operations, you need to choose which implementation of atomic operations that you want to use.

1. Open `FreeRTOSConfig.h` (p. 8) configuration file for edit.
2. For the ISA native atomic support implementation, define and set the `configUSE_ATOMIC_INSTRUCTION` variable to `1`.

   For the disabling global interrupt implementation, undefine or clear `configUSE_ATOMIC_INSTRUCTION`.

API reference

For a full API reference, see Atomic Operations C SDK API Reference.

Linear Containers library

The FreeRTOS Linear Containers library defines linear data structures, including lists and queues, for you to use when developing embedded applications.

API reference

For a full API reference, see Linear Containers C SDK API Reference.

Logging library

The FreeRTOS logging library enables FreeRTOS libraries to print messages to the log for debugging.

API reference

For a full API reference, see Logging C SDKAPI Reference.

Static Memory library

The FreeRTOS Static Memory library defines some functions for managing static buffers. With this library, you can use a Static Memory component to provide statically-allocated buffers instead of using dynamic memory allocation.

API reference

For a full API reference, see Static Memory C SDK API Reference.

Task Pool library

Overview

FreeRTOS supports task management with the FreeRTOS Task Pool library. The Task Pool library enables you to schedule background tasks, and allows safe, asynchronous task scheduling and cancellation. Using the Task Pool APIs, you can configure your application's tasks to optimize the trade-off between performance and memory footprint.

The Task Pool library is built on two main data structures: the Task Pool and Task Pool jobs.

Task Pool (`IotTaskPool_t`)

The Task Pool contains a dispatch queue that manages the job queue for execution, and manages the worker threads that execute jobs.

Task Pool jobs (`IotTaskPoolJob_t`)

Task Pool jobs can be executed as background jobs, or timed background jobs. Background jobs are started in First-In-First-Out order and have no time constraints. Timed jobs are scheduled for background execution according to a timer.

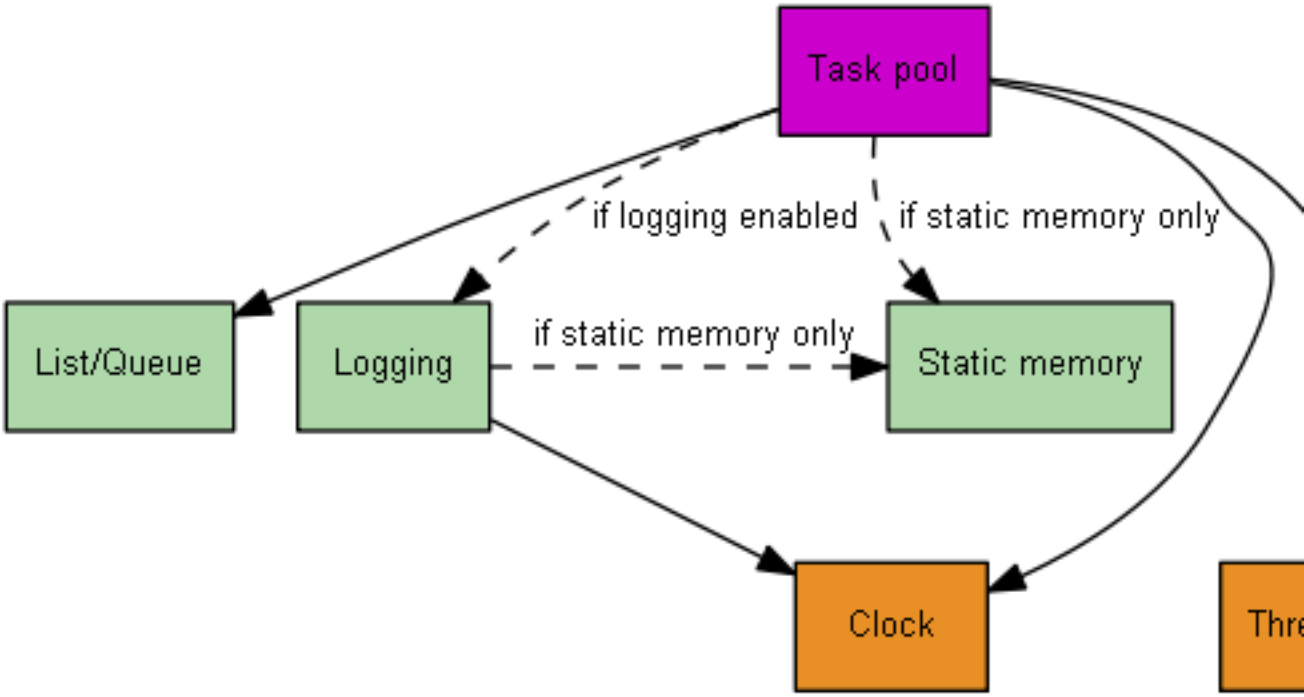
> **Note**
> Task Pool can only guarantee that a timed job will be executed after a timeout elapses, and not within a specific window of time.

Dependencies and requirements

The Task Pool library has the following dependencies:

- The Linear Containers (list/queue) library for maintaining the data structures for scheduled and in-progress task pool operations.
- The logging library (if `IOT_LOG_LEVEL_TASKPOOL` configuration setting is not `IOT_LOG_NONE`).

- A platform layer that provides an interface to the operating system for thread management, timers, clock functions, etc.



Features

Using the Task Pool library APIs, you can do the following:

- Schedule immediate and deferred jobs with the library's non-blocking API functions.
- Create statically and dynamically allocated jobs.
- Configure library settings to scale performance and footprint, based on your system's resources.
- Customize caching for low memory overhead when creating jobs dynamically.

Troubleshooting

The Task Pool library functions return error codes as `IotTaskPoolError_t` enumerated values. For more information about each error code, see the reference documentation for `IotTaskPoolError_t` enumerated data type in Task Pool C SDK API Reference.

Usage restrictions

The Task Pool pool library cannot be used from an interrupt service routine (ISR).

We strongly discourage task pool user callbacks that perform blocking operations, especially indefinite blocking operations. Long-standing blocking operations effectively steal a task pool thread, and create a potential for deadlock or starvation.

Initialization

An application needs to call `IotTaskPool_CreateSystemTaskPool` to initialize an instance of a system task pool, prior to using the task pool. The application needs to make sure that system-level task

pool is initialized early enough in the boot sequence, before any library uses the task pool, and before any application code posts a job to the task pool. Shortly after boot, the system initializes the single, system-level task pool for all libraries to share. After initialization, the task pool handle can be retrieved for use with the `IOT_SYSTEM_TASKPOOL` API.

Note
Calling `IotTaskPool_CreateSystemTaskPool` does not allocate memory to hold the task pool data structures and state, but it might allocate memory to hold the dependent entities and data structures, like the threads of the task pool.

API reference

For a full API reference, see Task Pool C SDK API Reference.

Example usage

Suppose that you need to schedule a recurring collection of AWS IoT Device Defender metrics, and you decide to use a timer to schedule the collection with calls to the MQTT connect, subscribe, and publish APIs. The following code defines a callback function for accepting AWS IoT Device Defender metrics across MQTT, with a disconnect callback that disconnects from the MQTT connection.

```
/* An example of a user context to pass to a callback through a task pool thread. */
typedef struct JobUserContext
{
    uint32_t counter;
} JobUserContext_t;

/* An example of a user callback to invoke through a task pool thread. */
static void ExecutionCb( IotTaskPool_t * pTaskPool, IotTaskPoolJob_t * pJob, void *
 context )
{
    ( void )pTaskPool;
    ( void )pJob;
    JobUserContext_t * pUserContext = ( JobUserContext_t * )context;
    pUserContext->counter++;
}

void TaskPoolExample( )
{
    JobUserContext_t userContext = { 0 };
    IotTaskPoolJob_t job;
    IotTaskPool_t * pTaskPool;
    IotTaskPoolError_t errorSchedule;

    /* Configure the task pool to hold at least two threads and three at the maximum. */
    /* Provide proper stack size and priority per the application needs. */
    const IotTaskPoolInfo_t tpInfo = { .minThreads = 2, .maxThreads = 3, .stackSize =
 512, .priority = 0 };

    /* Create a task pool. */
    IotTaskPool_Create( &tpInfo, &pTaskPool );

    /* Statically allocate one job, then schedule it. */
    IotTaskPool_CreateJob( &ExecutionCb, &userContext, &job );
    errorSchedule = IotTaskPool_Schedule( pTaskPool, &job, 0 );

    switch ( errorSchedule )
    {
    case IOT_TASKPOOL_SUCCESS:
        break;
    case IOT_TASKPOOL_BAD_PARAMETER:          // Invalid parameters, such as a NULL handle,
 can trigger this error.
```

```
    case IOT_TASKPOOL_ILLEGAL_OPERATION:      // Scheduling a job that was previously
 scheduled or destroyed could trigger this error.
    case IOT_TASKPOOL_NO_MEMORY:              // Scheduling a with flag
 #IOT_TASKPOOL_JOB_HIGH_PRIORITY could trigger this error.
    case IOT_TASKPOOL_SHUTDOWN_IN_PROGRESS:   // Scheduling a job after trying to destroy
 the task pool could trigger this error.
        // ASSERT
        break;
    default:
        // ASSERT*/
    }

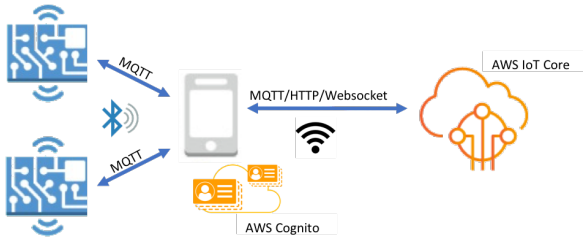
    /* ................................ */
    /*  ... Perform other operations ... */
    /* ................................ */

    IotTaskPool_Destroy( pTaskPool );
}
```

Bluetooth Low Energy library

Overview

FreeRTOS supports publishing and subscribing to MQTT topics over Bluetooth Low Energy through a proxy device, such as a mobile phone. With the FreeRTOS Bluetooth Low Energy 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 Bluetooth Low Energy. For more information about the mobile SDKs, see Mobile SDKs for FreeRTOS Bluetooth devices (p. 208).

The FreeRTOS Bluetooth Low Energy library includes services for configuring Wi-Fi networks, transferring large amounts of data, and providing network abstractions over Bluetooth Low Energy. The FreeRTOS Bluetooth Low Energy library also includes middleware and lower-level APIs for more direct control over your Bluetooth Low Energy stack.

Architecture

Three layers make up the FreeRTOS Bluetooth Low Energy library: services, middleware, and low-level wrappers.

Services

The FreeRTOS Bluetooth Low Energy services layer consists of four Generic Attributes (GATT) services that leverage the middleware APIs: Device Information, Wi-Fi Provisioning, Network Abstraction, and Large Object Transfer.

Device information

The Device Information service gathers information 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 interact with the service directly.

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

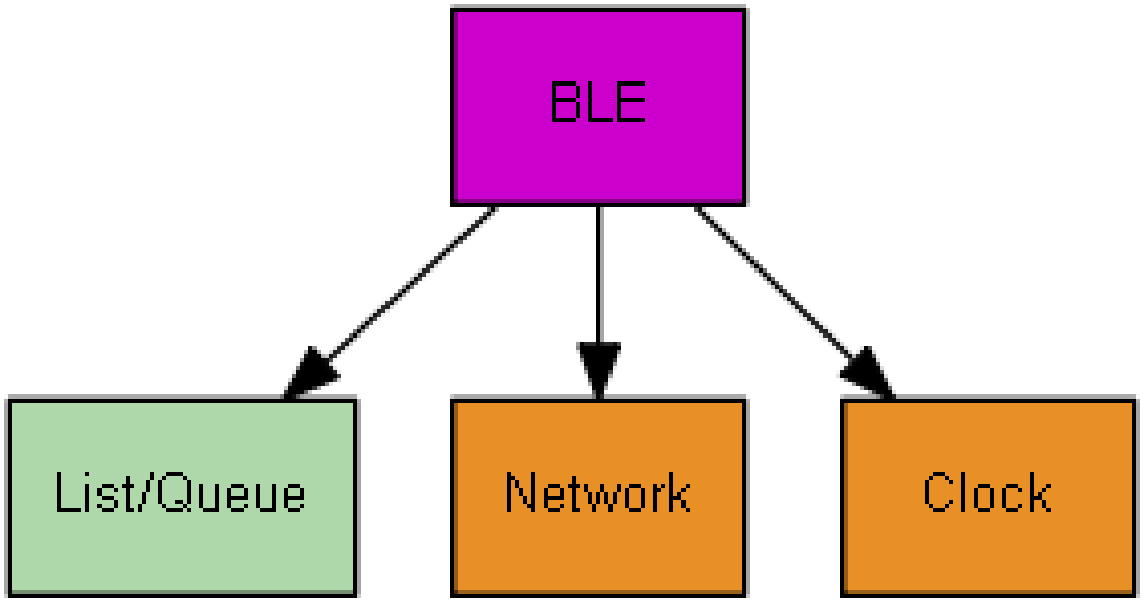
The low-level FreeRTOS Bluetooth Low Energy wrappers are an abstraction from the manufacturer's Bluetooth Low Energy stack. Low-level wrappers offer a common set of APIs for direct control over the hardware. The low-level APIs optimize RAM usage, but are limited in functionality.

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 (p. 195)
- 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 (p. 230)

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

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

1. Download the SDK from GitHub.
2. Open Android Studio, and import the `amazon-freertos-ble-android-sdk/amazonfreertossdk/` directory into your app project. The Android Studio User Guide has more information on using Android Studio.
3. In your app project's `gradle` file, add the following dependencies:

```
dependencies {
    implementation project(":amazonfreertossdk")
}
```

4. In your project's `settings.gradle` file, add `':amazonfreertossdk'`:

```
include ':app', ':amazonfreertossdk'
```

5. In your app project's `AndroidManifest.xml` file, add the following permissions:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
    <!-- initiate device discovery and manipulate bluetooth settings -->
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <!-- allow scan Bluetooth Low Energy -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- AWS Mobile SDK -->
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

For information about setting up and running the demo mobile application that is included with the SDK, see Prerequisites (p. 237) and FreeRTOS Bluetooth Low Energy Mobile SDK demo application (p. 239).

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 (p. 237) and FreeRTOS Bluetooth Low Energy Mobile SDK demo application (p. 239).

AWS IoT Device Defender library

Overview

AWS IoT Device Defender is an AWS IoT service that enables you to monitor connected devices to detect abnormal behavior and to mitigate security risks. With AWS IoT Device Defender, you can enforce consistent IoT configurations across your AWS IoT device fleet and respond quickly when devices are compromised.

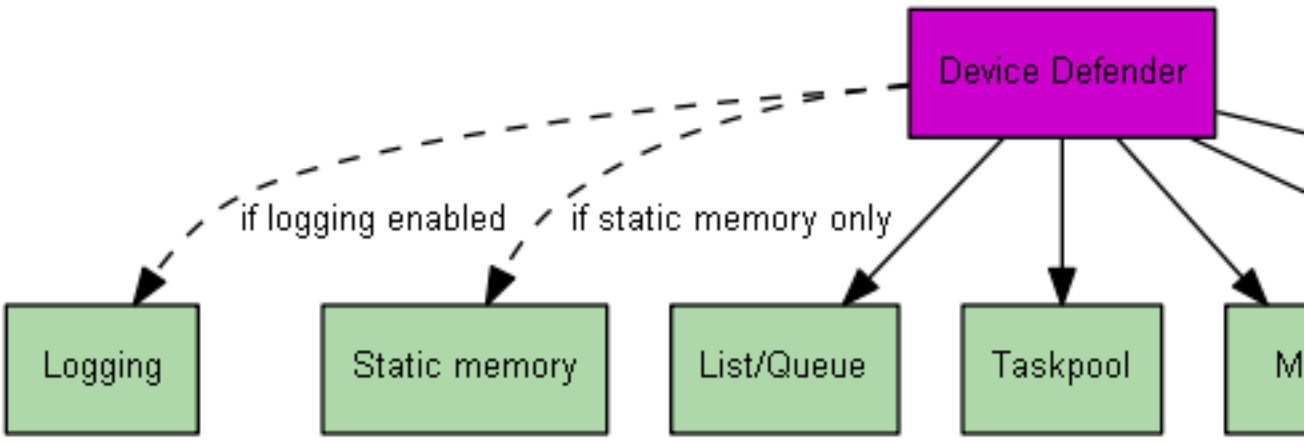
FreeRTOS provides a library that allows your FreeRTOS-based devices to work with AWS IoT Device Defender. You can download FreeRTOS with the Device Defender library from the FreeRTOS Console by adding the Device Defender library to your software configuration. You can also clone the FreeRTOS GitHub repository, which includes all FreeRTOS libraries. See the README.md file for instructions.

Note
The FreeRTOS AWS IoT Device Defender library only supports a subset of the device-side AWS IoT Device Defender metrics related to connection metrics. For more information, see Usage restrictions (p. 211).

Dependencies and requirements

The Device Defender library has the following dependencies:

- Linear Containers library (p. 195).
- Logging library (p. 196) (if the configuration parameter `AWS_IOT_LOG_LEVEL_DEFENDER` is not set to `IOT_LOG_NONE`).
- Static Memory library (p. 196) (if Static Memory only).
- A platform layer that provides an interface to the operating system for thread management, timers, clock functions, etc.
- Task Pool library (p. 196).
- MQTT library, version 2.0.0 (p. 216).



Troubleshooting

FreeRTOS Device Defender error codes

The Device Defender library returns error codes as positive values. For more information about each error code, see `AwsIotDefenderError_t` in the Device Defender C SDK API Reference.

FreeRTOS Device Defender events

The Device Defender library includes the `AwsIotDefenderCallback_t` callback function, which returns positive, enumerated values known as "events" that indicate success or failure. For more information about event types, see `AwsIotDefenderEventType_t` in the Device Defender C SDK API Reference.

Debugging FreeRTOS Device Defender

To enable the debugging for the Device Defender library, set the log level for Device Defender to debug mode in the global configuration file (p. 194):

```
#define  AWS_IOT_LOG_LEVEL_DEFENDER  IOT_LOG_DEBUG
```

For more information, see the Global Configuration File Reference.

Developer support

The Device Defender library includes the `AwsIotDefender_strerror` helper function, which returns a string that describes the error that you provide to the function:

```
const char * AwsIotDefender_strerror( AwsIotDefenderError_t error );
```

Usage restrictions

Although the AWS IoT Device Defender service supports both JSON and CBOR formats for data serialization, the FreeRTOS Device Defender library currently only supports CBOR, which is controlled by the configuration option `AWS_IOT_DEFENDER_FORMAT`.

Additionally, the FreeRTOS AWS IoT Device Defender library only supports a subset of device-side AWS IoT Device Defender metrics:

Long Name	Short Name	Parent Element	Description
`remote_addr`	`rad`	`connections`	Lists the remote address of a TCP connection.
`total`	`t`	`established_connections`	Lists the number of established TCP connections.

For example:

```
{
```

```
    "tcp_connections": {
        "established_connections": {
            "connections": [
                {
                    "remote_addr": "192.168.0.1:8000"
                },
                {
                    "remote_addr": "192.168.0.2:8000"
                }
            ],
            "total": 2
        }
    }
}
```

This JSON document is for example purposes only, as FreeRTOS Device Defender library does not support JSON-formatted metrics.

Initialization

The macro `AWS_IOT_SECURE_SOCKETS_METRICS_ENABLED` must be defined to enable the secure sockets metrics. Leaving this macro undefined could result in unpredictable behavior.

FreeRTOS Device Defender API

For a full API reference, see the Device Defender C SDK API Reference.

Example usage

For a full example of the Device Defender library in use, see AWS IoT Device Defender demo (p. 253).

AWS IoT Device Shadow library

Overview

The FreeRTOS Device Shadow APIs define functions to create, update, and delete AWS IoT Device Shadows. For more information about AWS IoT Device Shadows, see Device Shadow Service for AWS IoT. The Device Shadow service is accessed using the MQTT protocol. The FreeRTOS Device Shadow API works with the MQTT API to handle the details of working with the MQTT protocol.

Dependencies and requirements

To use AWS IoT Device Shadows with FreeRTOS, you need to register your device as an AWS IoT thing. Your thing must have a certificate with a policy that permits accessing the device's shadow. For more information, see AWS IoT Getting Started. For an example policy for FreeRTOS, see the AWS IoT Device Shadow demo application (p. 267).

An example client credentials header file is located at *`freertos`*`/demos/include/aws_clientcredential.h`. Make sure that you set values for the following constants in that header file:

`clientcredentialMQTT_BROKER_ENDPOINT`

Your AWS IoT endpoint.

`clientcredentialIOT_THING_NAME`

The name of your IoT thing.

`clientcredentialWIFI_SSID`

The SSID of your Wi-Fi network.

`clientcredentialWIFI_PASSWORD`

Your Wi-Fi password.

`clientcredentialWIFI_SECURITY`

The type of Wi-Fi security used by your network.

`keyCLIENT_CERTIFICATE_PEM`

The certificate PEM associated with your IoT thing.

`keyCLIENT_PRIVATE_KEY_PEM`

The private key PEM associated with your IoT thing.

This file is included in `aws_shadow_lightbulb_on_off.c` (the Device Shadow demo application (p. 267)).

If you are developing your own application, you need to include the `aws_client_credentials.h` header file in the application, and then pass the credentials as `MQTTAgentConnectParams` to `SHADOW_ClientConnect` to connect to AWS IoT over MQTT. Make sure that you specify your device's registered AWS IoT thing name for the `pucClientId` field of `MQTTAgentConnectParams`, or the Device Shadow client will not connect.

Before running the application, make sure the FreeRTOS MQTT library is installed on your device. For more information, see MQTT library (p. 216).

Also make sure that the MQTT buffers are large enough to contain the shadow JSON files. The maximum size for a device shadow document is 8 KB. All default settings for the device shadow API can be set in the `aws_shadow_config_defaults.h` file. You can modify any of these settings in the *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_shadow_config.h` file.

Important
The JSON format that you define for your Device Shadow tasks must include a `clientToken` field. The `clientToken` can take any unique value. For example, the `aws_shadow_lightbulb_on_off.c` demo application uses `token-`*`%d`*, where *`%d`* is the RTOS tick count at the time the JSON document is generated.
If the JSON format does not include a `clientToken` field, calls to SHADOW_Delete(), SHADOW_Get(), and SHADOW_Update() will timeout.

API reference

For a full API reference, see Device Shadow C SDK API Reference.

Example usage

1. Use the `SHADOW_ClientCreate` API to create a shadow client. For most applications, the only field to fill is `xCreateParams.xMQTTClientType = eDedicatedMQTTClient`.
2. Establish an MQTT connection by calling the `SHADOW_ClientConnect` API, passing the client handle returned by `SHADOW_ClientCreate`.
3. Call the `SHADOW_RegisterCallbacks` API to configure callbacks for shadow update, get, and delete.

After a connection is established, you can use the following APIs to work with the device shadow:

`SHADOW_Delete`

Delete the device shadow.

`SHADOW_Get`

Get the current device shadow.

`SHADOW_Update`

Update the device shadow.

Note
When you are done working with the device shadow, call `SHADOW_ClientDisconnect` to disconnect the shadow client and free system resources.

AWS IoT Greengrass Discovery library

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 MQTT 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.

MQTT library, version 2.0.0

Overview

You can use the FreeRTOS MQTT library to create applications that publish and subscribe to MQTT topics, as MQTT clients on a network. The FreeRTOS MQTT library implements the MQTT 3.1.1 standard for compatibility with the AWS IoT MQTT server. The library is also compatible with other MQTT servers.

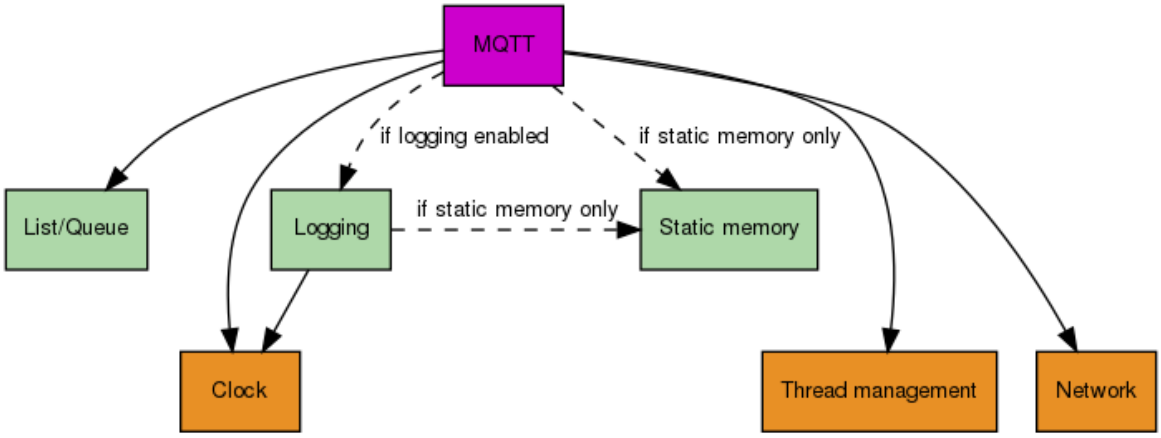
The source files for the FreeRTOS MQTT library are located in *`freertos`*`/.../mqtt`. These files implement version 2.0.0 of the FreeRTOS MQTT library. FreeRTOS also includes a backward-compatibility layer for version 1.0.0 of the FreeRTOS MQTT library. For information about FreeRTOS MQTT version 1.0.0, see MQTT library, version 1.0.0 (p. 218).

Dependencies and requirements

The FreeRTOS MQTT library has the following dependencies:

- Linear Containers library (p. 195).
- Logging library (p. 196) (if the configuration parameter `AWS_IOT_MQTT_LOG_LEVEL` is not set to `AWS_IOT_LOG_NONE`).
- Static Memory library (p. 196) (if Static Memory only).
- Task Pool library (p. 196).
- The platform layer that provides an interface to the operating system for thread management, clock functions, networking, and other platform-level functionality
- C standard library headers

The diagram below illustrates these dependencies.



Features

The FreeRTOS MQTT library has the following features:

- By default, the library has a fully asynchronous MQTT API. You can opt to use the library synchronously with the `AwsIotMqtt_Wait` function.
- The library is thread-aware and parallelizable for high throughput.
- The library features scalable performance and footprint. Use the configuration setting to tailor the library to a system's resources.

API reference

FreeRTOS and the AWS IoT Device SDK for Embedded C have the same MQTT v2.0.0 library APIs. For a full API reference, see MQTT (v2.0.0) C SDK API Reference.

Example usage

For example usage of the FreeRTOS MQTT library, see MQTT demo application defined in `iot_demo_mqtt.c`.

The MQTT demo demonstrates the subscribe-publish workflow of MQTT. After subscribing to multiple topic filters, the application publishes bursts of data to various topic names. As each message arrives, the demo publishes an acknowledgement message back to the MQTT server.

To use the MQTT demo, you must create a thing in AWS IoT, including a certificate and policy. For more information, see AWS IoT Getting Started.

Global demo configuration parameters

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.

MQTT demo configuration parameters

These configuration parameters apply to the MQTT demo.

`AWS_IOT_DEMO_MQTT_PUBLISH_BURST_SIZE`

The number of messages to publish in each burst.

`AWS_IOT_DEMO_MQTT_PUBLISH_BURST_COUNT`

The number of publish bursts in this demo.

MQTT library, version 1.0.0

Overview

FreeRTOS includes an open source MQTT client library that you can use to create applications that publish and subscribe to MQTT topics, as MQTT clients on a network.

Version 2.0.0 of the MQTT Library is available for FreeRTOS versions 201960.00 and later. This newer library is compatible with all transport types, meaning you can use it with Bluetooth Low Energy and TCP/IP. For more information, see MQTT library, version 2.0.0 (p. 216).

The FreeRTOS MQTT Agent

FreeRTOS also includes an open source daemon, called the FreeRTOS MQTT Agent, that manages the MQTT library for you. The MQTT Agent provides a simple interface to connect, publish, and subscribe to MQTT topics with the underlying MQTT library.

The MQTT Agent runs in a separate FreeRTOS task and automatically sends regular keep-alive messages, as documented by the MQTT protocol specification. All the MQTT APIs are blocking and take a timeout parameter, which is the maximum amount of time the API waits for the corresponding operation to complete. If the operation does not complete in the provided time, the API returns timeout error code.

Dependencies and requirements

The FreeRTOS MQTT library uses the Secure Sockets library (p. 226) and the FreeRTOS Buffer Pool library. If the MQTT Agent connects to a secure MQTT broker, the library also uses the Transport Layer Security (p. 230).

Features

Callback

You can specify an optional callback that is invoked whenever the MQTT Agent is disconnected from the broker or whenever a publish message is received from the broker. The received publish message is stored in a buffer taken from the central buffer pool. This message is passed to the callback. This callback runs in the context of the MQTT task and therefore must be quick. If you

need to do longer processing, you must take the ownership of the buffer by returning `pdTRUE` from the callback. You must then return the buffer back to the pool whenever you are done by calling `FreeRTOS_Agent_ReturnBuffer`.

Subscription management

Subscription management enables you to register a callback per subscription filter. You supply this callback while subscribing. It is invoked whenever a publish message received on a topic matches the subscribed topic filter. The buffer ownership works the same way as described in the case of generic callback.

MQTT task wakeup

MQTT task wakeup wakes up whenever the user calls an API to perform any operation or whenever a publish message is received from the broker. This asynchronous wakeup upon receipt of a publish message is possible on platforms that are capable of informing the host MCU about the data received on a connected socket. Platforms that do not have this capability require the MQTT task to continuously poll for the received data on the connected socket. To ensure the delay between receiving a publish message and invoking the callback is minimal, the `mqttconfigMQTT_TASK_MAX_BLOCK_TICKS` macro controls the maximum time an MQTT task can remain blocked. This value must be short for the platforms that lack the capability to inform the host MCU about received data on a connected socket.

Major configurations

These flags can be specified during the MQTT connection request:

- `mqttconfigKEEP_ALIVE_ACTUAL_INTERVAL_TICKS`: The frequency of the keep-alive messages sent.
- `mqttconfigENABLE_SUBSCRIPTION_MANAGEMENT`: Enable subscription management.
- `mqttconfigMAX_BROKERS`: Maximum number of simultaneous MQTT clients.
- `mqttconfigMQTT_TASK_STACK_DEPTH`: The task stack depth.
- `mqttconfigMQTT_TASK_PRIORITY`: The priority of the MQTT task.
- `mqttconfigRX_BUFFER_SIZE`: Length of the buffer used to receive data.
- `mqttagentURL_IS_IP_ADDRESS`: Set this bit in `xFlags` if the provided URL is an IP address.
- `mqttagentREQUIRE_TLS`: Set this bit in `xFlags` to use TLS.
- `mqttagentUSE_AWS_IOT_ALPN_443`: Set this bit in `xFlags` to use AWS IoT support for MQTT over TLS port 443.

For more information about ALPN, see the AWS IoT Protocols in the AWS IoT Developer Guide and the MQTT with TLS Client Authentication on Port 443: Why It Is Useful and How It Works blog post on the Internet of Things on AWS blog.

Optimization

Processing received packets without delay

The task that implements the MQTT Agent spends most of its time in the Blocked state (so not using any CPU cycles) waiting for events to process. MQTT throughput is maximized by unblocking the Agent task as soon as an MQTT packet is received from the network. If that is done the received packet is processed at the earliest opportunity. If that is not done the received packet will not be processed until the MQTT Agent leaves the Blocked state for another reason.

The MQTT Agent is removed from the Blocked state by the execution of a callback that is installed by the MQTT Agent calling SOCKETS_SetSockOpt() with the lOptionName parameter set to SOCKETS_SO_WAKEUP_CALLBACK. Links to the secure sockets documentation are needed here. If you are using the FreeRTOS+TCP TCP/IP stack the callback is executed at the correct time provided ipconfigSOCKET_HAS_USER_WAKE_CALLBACK is set to 1 in FreeRTOSIPConfig.h (which is the TCP/IP stack's configuration file). If you are not using the FreeRTOS+TCP TCP/IP stack then the secure sockets ensure this functionality is included in your implementation of the secure sockets abstraction layer for the stack in use.

If the TCP/IP stack cannot unblock the MQTT Agent as soon as data is received then the maximum time between a packet being received and the packet being processed is set by the mqttconfigMQTT_TASK_MAX_BLOCK_TICKS constant.

Minimizing RAM consumption

The following configuration constants directly affect the amount of RAM required by the MQTT Agent:

- mqttconfigMQTT_TASK_STACK_DEPTH
- mqttconfigMQTT_TASK_STACK_DEPTH
- mqttconfigMAX_BROKERS
- mqttconfigMAX_PARALLEL_OPS
- mqttconfigRX_BUFFER_SIZE

You should set these constants to the minimum values possible.

Requirements and usage restrictions

The MQTT Agent task is created using the xTaskCreateStatic() API function - so the task's stack and control block are statically allocated at compile time. That ensures the MQTT Agent can be used in applications that do not allow dynamic memory allocation, but does mean there is a dependency on configSUPPORT_STATIC_ALLOCATION being set to 1 in `FreeRTOSConfig.h` (p. 8).

he MQTT Agent uses the FreeRTOS direct to task notification feature. Calling an MQTT Agent API function may change the calling task's notification value and state.

MQTT packets are stored in buffers provided by the Buffer Pool module. It is highly recommended to ensure the number of buffers in the pool is at least double the number of MQTT transactions that will be in progress at any one time.

Developer support

`mqttconfigASSERT`

mqttconfigASSERT() is equivalent to, and used in exactly the same way as, the FreeRTOS configASSERT() macro. If you want assert statements in the MQTT Agent then define mqttconfigASSERT(). If you do not want assert statements in the MQTT Agent then leave mqttconfigASSERT() undefined. If you define mqttconfigASSERT() to call the FreeRTOS configASSERT(), as shown below, then the MQTT Agent will only include assert statements if the FreeRTOS configASSERT() is defined.

```
#define mqttconfigASSERT( x ) configASSERT( x )
```

`mqttconfigENABLE_DEBUG_LOGS`

Set `mqttconfigENABLE_DEBUG_LOGS` to **1** to print debug logs via calls to vLoggingPrintf().

Initialization

Both the MQTT Agent and its dependent libraries must be initialized, as shown below, before attempting MQTT communication. Initialize the libraries after a network connection is established.

```
BaseType_t SYSTEM_Init() { BaseType_t xResult = pdPASS; /* The bufferpool libraries
 provides the buffers use to store MQTT packets.*/
                        xResult = BUFFERPOOL_Init();
                        if( xResult == pdPASS ) { /* Create the MQTT Agent task. */
                        xResult = MQTT_AGENT_Init(); }
                        if( xResult == pdPASS ) { /* Initialize the secure sockets
 abstraction layer.*/
                        xResult = SOCKETS_Init(); }
                        return xResult; }
```

API reference

For a full API reference, see MQTT (v1.0.0) Library API Reference and MQTT (v1) Agent API Reference.

Porting

The Secure Sockets abstraction layer that the MQTT Agent calls must be ported to specific architectures. For more information, see Porting the Secure Sockets Library in the FreeRTOS Porting Guide.

HTTPS client library

Overview

You can use the FreeRTOS HTTPS Client library to create applications that interact with an HTTP server to send HTTP requests and receive HTTP responses over TLS. The FreeRTOS HTTPS Client library implements the HTTP/1.1 standard over TLS.

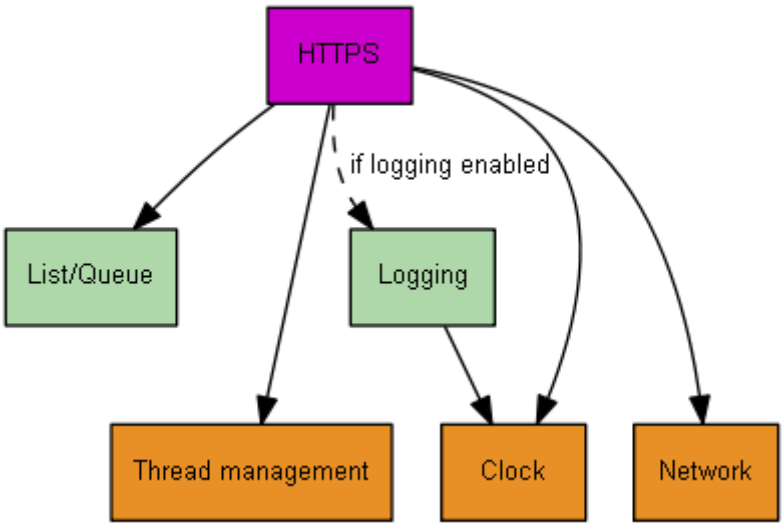
The source files for the FreeRTOS HTTPS Client library are located in *freertos*`/libraries/csdk/standard/https`.

Dependencies and requirements

The FreeRTOS HTTPS Client library has the following dependencies:

- Linear Containers library (p. 195).
- Logging library (p. 196) (if the configuration parameter `AWS_IOT_HTTPS_LOG_LEVEL` is not set to `AWS_IOT_LOG_NONE`).
- Static Memory library (p. 196) (if Static Memory only).
- Task Pool library (p. 196).
- The platform layer that provides an interface to the operating system for thread management, clock functions, networking, and other platform-level functionality.
- C standard library headers.

The diagram below illustrates these dependencies.



Features

The FreeRTOS HTTPS Client library has the following features:

- Both fully asynchronous and synchronous (blocking) API functions.
- Application managed memory for internal context and HTTP formatted headers.
- Thread-aware and parallelized connections.

API reference

For a full API reference, see the HTTPS Client API Reference.

OTA Agent library

Overview

The Over-The-Air (OTA) Agent 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.

For more information about setting up OTA updates with FreeRTOS, see FreeRTOS Over-the-Air Updates (p. 8).

Features

Here is the complete OTA Agent interface:

`OTA_AgentInit`

Initializes the OTA Agent. The caller provides messaging protocol context, an optional callback, and a timeout.

`OTA_AgentShutdown`

Cleans up resources after using the OTA Agent.

`OTA_GetAgentState`

Gets the current state of the OTA Agent.

`OTA_ActivateNewImage`

Activates the newest microcontroller firmware image received through OTA. (The detailed job status should now be self-test.)

`OTA_SetImageState`

Sets the validation state of the currently running microcontroller firmware image (testing, accepted or rejected).

`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.

API reference

For more information, see the OTA Agent API Reference.

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 MQTT broker. For more information, see MQTT library, version 1.0.0 (p. 218).
2. Initialize the OTA Agent by calling `OTA_AgentInit`. Your application may define a custom OTA callback function or use the default callback by specifying a NULL callback function pointer. You must also supply an initialization timeout.

   The callback implements application-specific logic that executes after completing an OTA update job. The timeout defines how long to wait for the initialization to complete.
3. If `OTA_AgentInit` timed out before the Agent was ready, you can call `OTA_GetAgentState` to confirm that the Agent is initialized and operating as expected.
4. When the OTA update is complete, FreeRTOS calls the job completion callback with one of the following events: `accepted`, `rejected`, or `self test`.
5. 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.
6. 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.

Public Key Cryptography Standard (PKCS) #11 library

Overview

Public Key Cryptography Standard #11 (PKCS#11) is a cryptographic API that abstracts key storage, get/set properties for cryptographic objects, and session semantics. See `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 PKCS#11 reference implementation supports 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 steps are written with the assumption that you have used the **aws configure** command to configure the AWS CLI.

Creating 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.

```
{
  "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 PKCS #11 library to your platform, see Porting the PKCS #11 Library in the FreeRTOS Porting Guide.

Secure Sockets library

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.

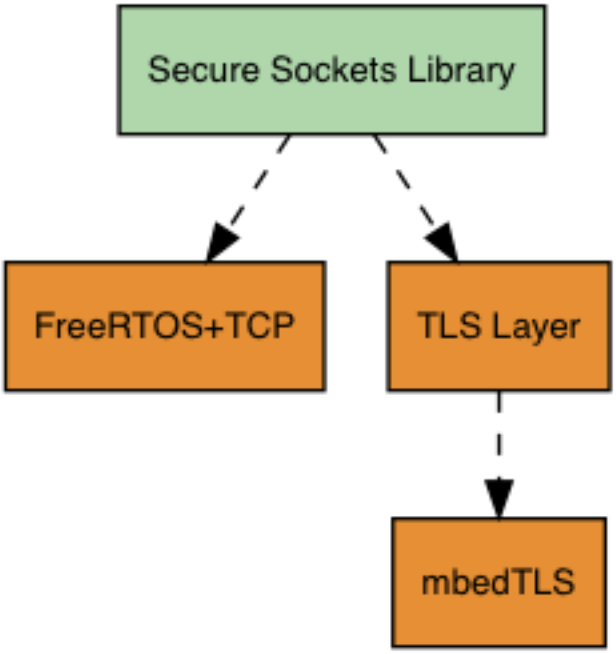
Note
Currently, only client APIs are supported for FreeRTOS Secure Sockets.

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 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 (p. 230).



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 MQTT library, version 1.0.0 (p. 218), 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.

Only client APIs are supported by the FreeRTOS Secure Sockets library. Server APIs, including `Bind`, `Accept`, and `Listen`, are not 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 (p. 269).

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 Public Key Cryptography Standard (PKCS) #11 library (p. 224)
- The Transport Layer Security (p. 230)

For more information about porting, see Porting the Secure Sockets Library in the FreeRTOS Porting Guide.

Transport Layer Security

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

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, and WPA2 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, and WPA2 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`

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 (p. 233) and API reference (p. 233).

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 (p. 233).

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 (p. 233) and API reference (p. 233).

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.

Common I/O

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.

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. If you use the FreeRTOS console, only the target platform you choose has a subdirectory under `demos`.

Before you try the demo applications, we recommend that you complete the tutorial in Getting Started with FreeRTOS (p. 71). It shows you how to set up and run the Hello World MQTT demo.

Running the FreeRTOS demos

The following topics show you how to set up and run the FreeRTOS demos:

- Bluetooth Low Energy demo applications (p. 236)
- Demo bootloader for the Microchip Curiosity PIC32MZEF (p. 248)
- AWS IoT Device Defender demo (p. 253)
- AWS IoT Greengrass discovery demo application (p. 254)
- Over-the-air updates demo application (p. 257)
- Secure Sockets echo client demo (p. 269)
- AWS IoT Device Shadow demo application (p. 267)

The `DEMO_RUNNER_RunDemos()` function, located in *`freertos`*`/demos/demo_runner/iot_demo_runner.c`, initializes a detached thread on which a single demo application runs. By default, `DEMO_RUNNER_RunDemos()` only calls the starts the Hello World MQTT demo. Depending on the configuration you selected when you downloaded FreeRTOS, and depending on where you downloaded FreeRTOS, the other example runner functions might start by default. To enable a demo application, open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, and define the demo that you want to run.

Note
Be aware that not all combinations of examples work together. Depending on the combination, the software might fail to execute 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

Overview

FreeRTOS Bluetooth Low Energy includes three demo applications:

MQTT over Bluetooth Low Energy (p. 243) demo

This application demonstrates how to use the MQTT over Bluetooth Low Energy service.

Wi-Fi provisioning (p. 245) demo

This application demonstrates how to use the Bluetooth Low Energy Wi-Fi Provisioning service.

Generic Attributes Server (p. 247) demo

This application demonstrates how to use the FreeRTOS Bluetooth Low Energy middleware APIs to create a simple GATT server.

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 (p. 209) or the Android SDK for FreeRTOS Bluetooth devices (p. 209).

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:

```
{
   "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 (p. 199) 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 (p. 243) 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 in the 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 you the iOS SDK for FreeRTOS Bluetooth devices (p. 209) 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 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.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource":"arn:aws:iot:aws-region:aws-account-id:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
        },
        {
             "Effect": "Allow",
             "Action": "iot:Subscribe",
             "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
        },
        {
             "Effect": "Allow",
             "Action": "iot:Receive",
             "Resource": "arn:aws:iot:aws-region:aws-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 (p. 209) 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 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.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Connect",
            "Resource":"arn:aws:iot:aws-region:aws-account-id:*"
        },
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
        },
        {
             "Effect": "Allow",
             "Action": "iot:Subscribe",
             "Resource": "arn:aws:iot:aws-region:aws-account-id:*"
        },
        {
             "Effect": "Allow",
             "Action": "iot:Receive",
             "Resource": "arn:aws:iot:aws-region:aws-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.

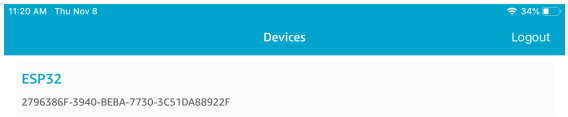
To discover and establish secure connections with your microcontroller over Bluetooth Low Energy

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 (p. 79).
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:

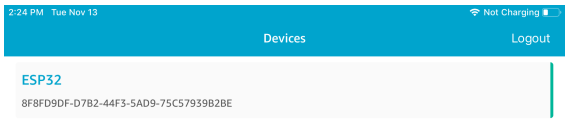
   ```
   $ ./gradlew installDebug
   ```

4. Confirm that your microcontroller appears under **Devices** on the Bluetooth Low Energy Mobile SDK demo app.

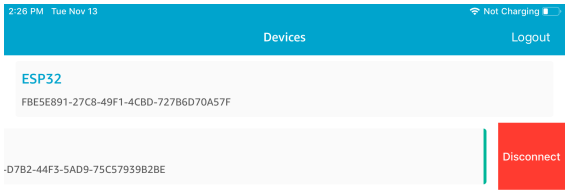


   Note
   All devices with FreeRTOS and the device information service (*`freertos`*`/.../device_information`) that are in range appear in the list.

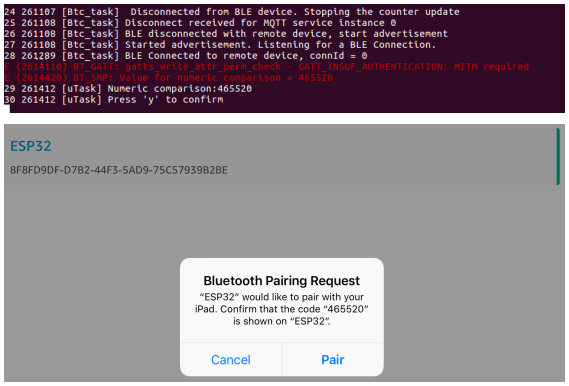
5. Choose your microcontroller from the list of devices. The application establishes a connection with the board, and a green line appears next to the connected device.



You can disconnect from your microntroller by dragging the line to the left.



6. If prompted, pair your microcontroller and mobile device.



If the code for numeric comparison is the same on both devices, pair the devices.

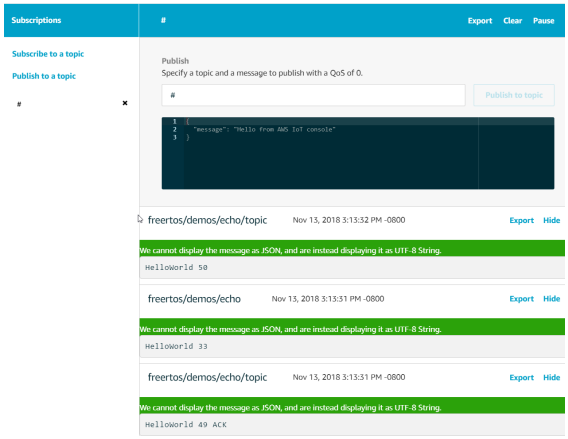
Note
The Bluetooth Low Energy Mobile SDK demo application uses Amazon Cognito for user authentication. Make sure that you have set up a Amazon Cognito user and identity pools, and that you have attached IAM policies to authenticated identities.

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** to open the MQTT client.
3. In **Subscription topic**, enter **`iotdemo/#`**, and then choose **Subscribe to topic**.



You can run the MQTT demo over a Bluetooth Low Energy or Wi-Fi connection. The configuration of the Network Manager (p. 239) determines which connection type is used.

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.

If you use Wi-Fi, the demo is the same as the MQTT Hello World demo project located at `demos/mqtt/aws_hello_world.c`. That demo is used in most of the Getting Started with FreeRTOS demo projects.

To enable the demo over Bluetooth Low Energy or Wi-Fi

Open `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, and define `CONFIG_MQTT_DEMO_ENABLED`.

- Open *`freertos`*`/aws_demos/config_files/aws_demo_config.h`, and configure your network type.

  Configurations include:

  ```
  #define democonfigNETWORK_TYPES ( AWSIOT_NETWORK_TYPE_BLE )
  ```

  Run when Bluetooth Low Energy gets a connection.

  ```
  #define democonfigNETWORK_TYPES ( AWSIOT_NETWORK_TYPE_WIFI)
  ```

  Run when WIFI connects to AP.

  ```
  #define democonfigNETWORK_TYPES ( AWSIOT_NETWORK_TYPE_WIFI|
  AWSIOT_NETWORK_TYPE_BLE)
  ```

  Connects to the first available network type. If both types are available, Wi-Fi is used.

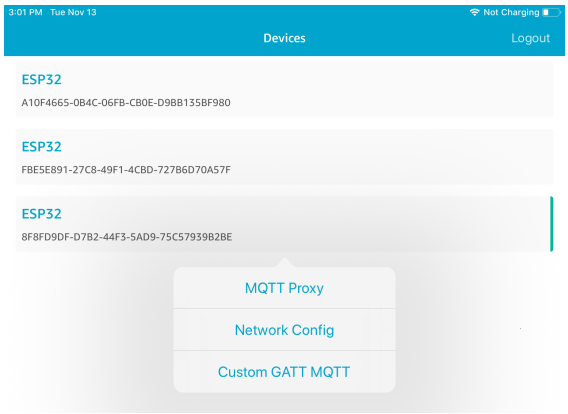
To run the demo

If the Network Manager is configured for Wi-Fi only, simply build and run the demo project on your board.

If the Network Manager is configured for Bluetooth Low Energy, do the following:

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 (p. 239).

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 `iotdemo/#` 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

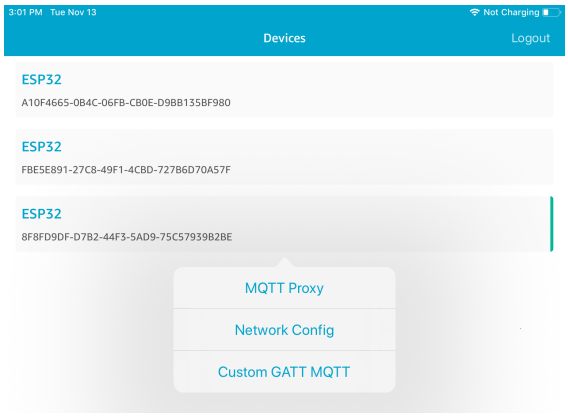
1. Enable the Wi-Fi Provisioning service. Open `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/iot_ble_config.h`, and set `#define IOT_BLE_ENABLE_WIFI_PROVISIONING` to `1` (where the *`vendor`* is the name of the vendor and the *`board`* is the name of the board that you are using to run the demos).

   Note
   The Wi-Fi Provisioning service is disabled by default.

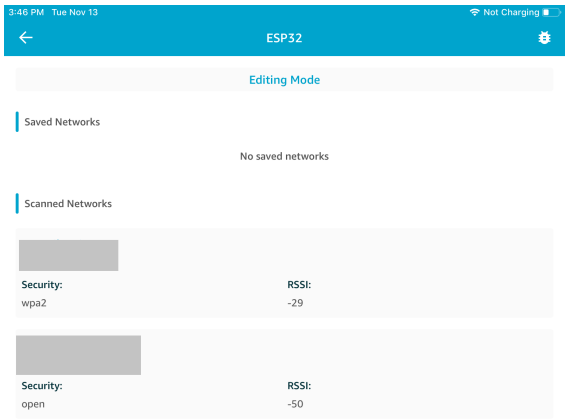
2. Configure the Network Manager (p. 239) to enable both Bluetooth Low Energy and Wi-Fi.

To run the demo

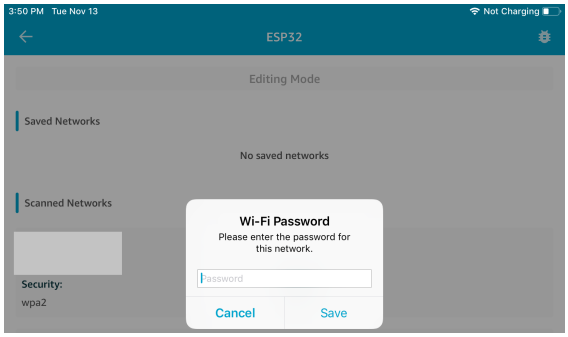
1. Build and run the demo project on your microcontroller.
2. Make sure that you have paired your microntroller and your mobile device using the FreeRTOS Bluetooth Low Energy Mobile SDK demo application (p. 239).
3. From the **Devices** list in the demo mobile app, choose your microcontroller, and then choose **Network Config** to open the network configuration settings.



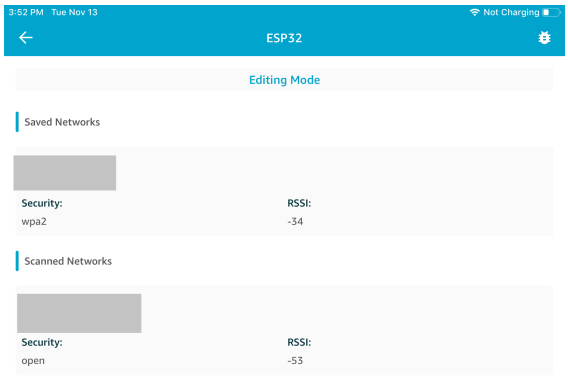
4. After you choose **Network Config** for your board, the microcontroller sends a list of the networks in the vicinity to the mobile device. Available Wi-Fi networks appear in a list under **Scanned Networks**.



From the **Scanned Networks** list, choose your network, and then enter the SSID and password, if required.

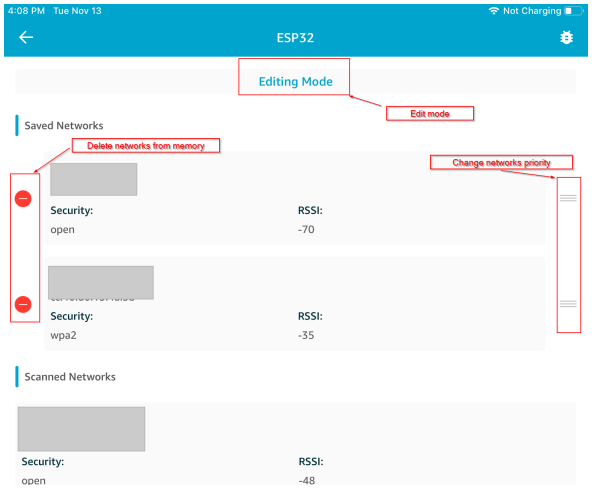


The micrcontroller connects to and saves the network. The network appears under **Saved Networks**.



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 (p. 239).

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

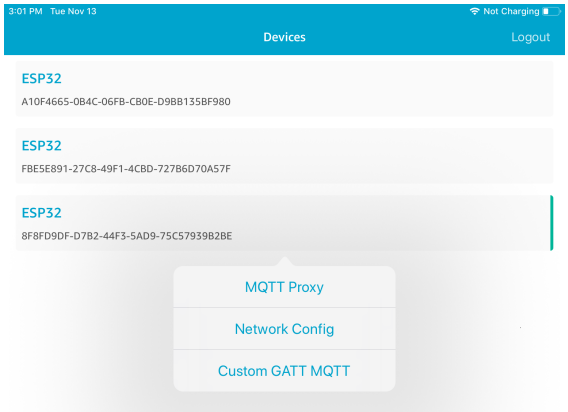
1. Enable the Bluetooth Low Energy GATT demo. In `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/iot_ble_config.h` (where the *`vendor`* is the name of the vendor and the *`board`* is the name of the board that you are using to run the demos), add `#define IOT_BLE_ADD_CUSTOM_SERVICES ( 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_MQTT_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 (p. 239).
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 `iotdemo/#` MQTT topic.

   After you enable the MQTT proxy, Hello World and incrementing counter messages appear on the `iotdemo/#` topic.

Demo bootloader for the Microchip Curiosity PIC32MZEF

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.

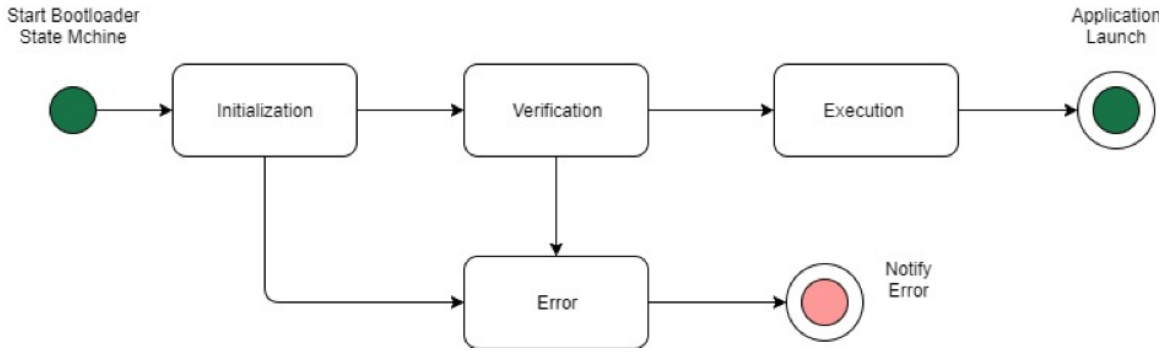
The firmware verification includes verifying the authenticity and integrity of the new firmware received over the air. The bootloader verifies the cryptographic signature of the application before booting. The demo uses elliptic-curve digital signature algorithm (ECDSA) over SHA-256. The utilities provided 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.

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

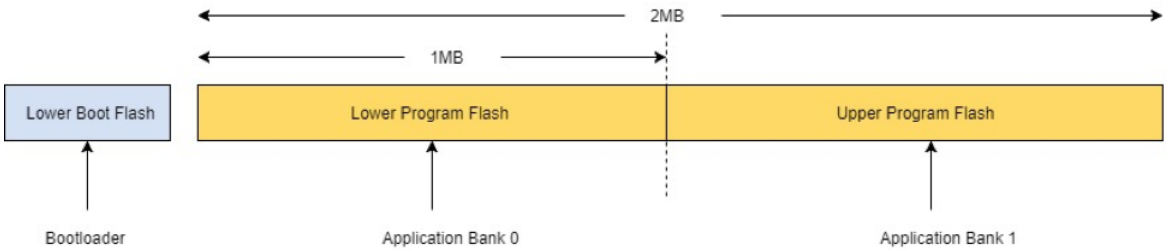
If the configuration option to launch the default image is enabled, the bootloader launches the application from a default execution address. This option must be disabled except while debugging.

Bootloader Error State

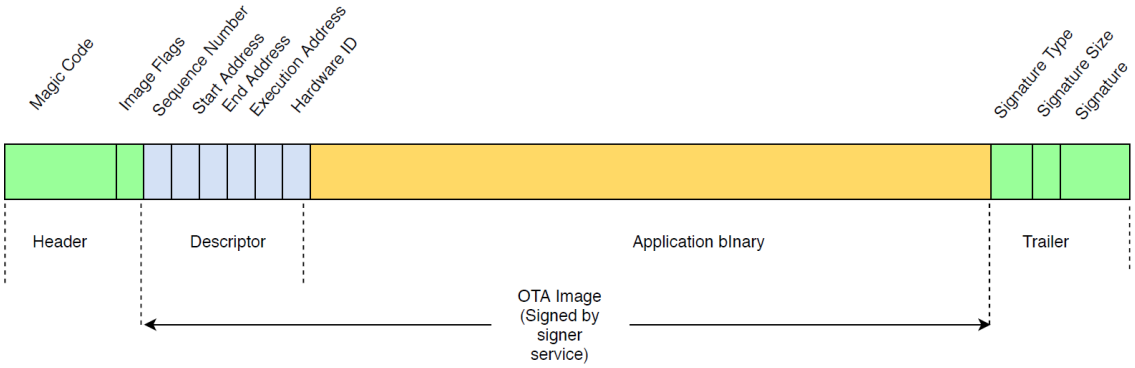
The bootloader is in an error state and no valid images are present on the device. The bootloader must notify the user. The default implementation sends a log message to the console and fast-blinks the LED on the board indefinitely.

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

FreeRTOS includes a single-threaded demo application that collects some AWS IoT Device Defender metrics for a device and publishes them to an MQTT topic. This demo is defined in *`freertos`*`/demos/defender/aws_iot_defender_demo.c`.

Before you can run the Device Defender demo, you must complete the getting started First steps (p. 72) to set up AWS IoT and FreeRTOS so your device can communicate with the AWS Cloud.

Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_MQTT_DEMO_ENABLED`, and define `CONFIG_DEFENDER_DEMO_ENABLED`.

When you build, flash, and run FreeRTOS on your device with the Device Defender demo enabled, the following output should appear:

```
12 343 [iot_thread] [INFO ][DEMO][343] ----Device Defender Demo Start----
13 343 [iot_thread] [INFO ][MQTT][343] MQTT library successfully initialized.
14 343 [iot_thread] [INFO ][Defender][343] Metrics are successfully updated.
15 343 [iot_thread] [INFO ][Defender][343] Period has been set to 300 seconds successfully.
```

```
16 343 [iot_thread] [INFO ][DEMO][343] Defender Thing Name is Thing-1 (length 7).
17 711 [iot_thread] [INFO ][MQTT][711] Establishing new MQTT connection.
18 711 [iot_thread] [INFO ][MQTT][711] Anonymous metrics (SDK language, SDK version) will
 be provided to AWS IoT. Recompile with AWS_IOT_MQTT_ENABLE_METRICS set to 0 to disable.
19 711 [iot_thread] [INFO ][MQTT][711] (MQTT connection 00530B30, CONNECT operation
 00530CC0) Waiting for operation completion.
20 771 [iot_thread] [INFO ][MQTT][771] (MQTT connection 00530B30, CONNECT operation
 00530CC0) Wait complete with result SUCCESS.
21 771 [iot_thread] [INFO ][MQTT][771] New MQTT connection 0203FA0C established.
22 771 [iot_thread] [INFO ][MQTT][771] (MQTT connection 00530B30) SUBSCRIBE operation
 scheduled.
23 771 [iot_thread] [INFO ][MQTT][771] (MQTT connection 00530B30, SUBSCRIBE operation
 00530E30) Waiting for operation completion.
24 811 [iot_thread] [INFO ][MQTT][811] (MQTT connection 00530B30, SUBSCRIBE operation
 00530E30) Wait complete with result SUCCESS.
25 811 [iot_thread] [INFO ][Defender][811] Defender agent has successfully started.
26 812 [iot_thread] [INFO ][MQTT][812] (MQTT connection 00530B30) MQTT PUBLISH operation
 queued.
27 932 [iot_thread] [INFO ][Defender][932] Metrics report was accepted by defender service.
28 932 [iot_thread] [INFO ][DEMO][932] User's callback is invoked on event: Defender
 Metrics accepted.
29 932 [iot_thread] [INFO ][DEMO][932] Published metrics report.
30 932 [iot_thread] [INFO ][DEMO][932] Received MQTT message.
31 8811 [iot_thread] [INFO ][Defender][8811] Unsubscribing from MQTT topics
32 8811 [iot_thread] [INFO ][MQTT][8811] (MQTT connection 00530B30) UNSUBSCRIBE operation
 scheduled.
33 8811 [iot_thread] [INFO ][MQTT][8811] (MQTT connection 00530B30, UNSUBSCRIBE operation
 00530F40) Waiting for operation completion.
34 8891 [iot_thread] [INFO ][MQTT][8891] (MQTT connection 00530B30, UNSUBSCRIBE operation
 00530F40) Wait complete with result SUCCESS.
35 8891 [iot_thread] [INFO ][Defender][8891] Defender agent has stopped.
35 8891 [iot_thread] [INFO ][MQTT][8891] (MQTT connection 00530B30) Disconnecting
 connection.
37 8892 [iot_thread] [INFO ][MQTT][8892] (MQTT connection 00530B30, DISCONNECT operation
 00530CE0) Waiting for operation completion.
38 8892 [iot_thread] [INFO ][MQTT][8892] (MQTT connection 00530B30, DISCONNECT operation
 00530CE0) Wait complete with result SUCCESS.
39 8892 [iot_thread] [INFO ][MQTT][8892] (MQTT connection 00530B30) Connection
 disconnected.
40 8893 [iot_thread] [INFO ][MQTT][8893] (MQTT connection 00530B30) Network connection
 closed.
41 8931 [iot_thread] [INFO ][MQTT][8931] (MQTT connection 00530B30) Network connection
 destroyed.
42 8931 [iot_thread] [INFO ][MQTT][8931] MQTT library cleanup done.
43 8931 [iot_thread] [INFO ][DEMO][8931] ----Device Defender Demo End. Status:
 SUCCESS----.
```

AWS IoT Greengrass discovery demo application

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 (p. 72). 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**.

You can use the **Quick Connect** workflow in the FreeRTOS console to quickly connect your board to AWS IoT and run the demo. FreeRTOS configurations are currently not available for the following boards:

- Cypress CYW943907AEVAL1F Development Kit
- Cypress CYW954907AEVAL1F Development Kit
- Espressif ESP-WROVER-KIT
- Espressif ESP32-DevKitC
- Nordic nRF52840-DK

You can also connect your board to AWS IoT and configure your FreeRTOS demo manually.

1. Registering your MCU board with AWS IoT (p. 73)

   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 (p. 75)

   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 (p. 75).

   Open *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h, comment out `#define CONFIG_MQTT_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 (p. 86).

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**, 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	-

Type	Protocol	Port range	Source	Description - optional
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

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.

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/aws_iot_ota_update_demo.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 (`vRunOTAUpdateDemo()`).
3. Creates an MQTT client (`prxCreateNetworkConnection()`).
4. Connects to the AWS IoT MQTT broker (`IotMqtt_Connect()`).
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 (`xOTAConnectionCtx.pvControlClient = xConnection.xMqttConnection;`).

Before you can use OTA updates, complete all prerequisites in the FreeRTOS Over-the-Air Updates (p. 8)

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 (p. 260)
- Microchip Curiosity PIC32MZEF (p. 262)
- Espressif ESP32 (p. 266)

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.

```
313 267848 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
314 268733 [OTA Task] [OTA] Set job doc parameter [ jobId:
 fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
315 268734 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
316 268734 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
317 268734 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
318 268735 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
319 268735 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
320 268735 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
321 268737 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa:
 Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
322 268737 [OTA Task] [OTA] Job was accepted. Attempting to start transfer.
323 268737 [OTA Task] Sending command to MQTT task.
324 268737 [MQTT] Received message 50000 from queue.
325 268848 [OTA] [OTA] Queued: 2   Processed: 1   Dropped: 0
326 269039 [MQTT] MQTT Subscribe was accepted. Subscribed.
327 269039 [MQTT] Notifying task.
328 269040 [OTA Task] Command sent to MQTT task passed.
329 269041 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/streams/327

330 269848 [OTA] [OTA] Queued: 2   Processed: 1   Dropped: 0
... // Output removed for brevity
346 284909 [OTA Task] [OTA] file token: 74594452
.. // Output removed for brevity
363 301327 [OTA Task] [OTA] file ready for access.
364 301327 [OTA Task] [OTA] Returned buffer to MQTT Client.
365 301328 [OTA Task] Sending command to MQTT task.
366 301328 [MQTT] Received message 60000 from queue.
367 301328 [MQTT] Notifying task.
368 301329 [OTA Task] Command sent to MQTT task passed.
369 301329 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/
streams/327/get
370 301632 [OTA Task] [OTA] Received file block 0, size 1024
371 301647 [OTA Task] [OTA] Remaining: 127
... // Output removed for brevity
508 304622 [OTA Task] Sending command to MQTT task.
509 304622 [MQTT] Received message 70000 from queue.
510 304622 [MQTT] Notifying task.
511 304623 [OTA Task] Command sent to MQTT task passed.
512 304623 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/
streams/327/get
513 304860 [OTA] [OTA] Queued: 47   Processed: 47   Dropped: 83
514 304926 [OTA Task] [OTA] Received file block 4, size 1024
515 304941 [OTA Task] [OTA] Remaining: 82
... // Output removed for brevity
797 315047 [MQTT] MQTT Publish was successful.
798 315048 [MQTT] Notifying task.
799 315048 [OTA Task] Command sent to MQTT task passed.
800 315049 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/
jobs/fe18c7ec_8c31_4438_b0b9_ad55acd9561801 315049 [OTA Task] Sending command to MQTT task.
```

```
802 315049 [MQTT] Received message d0000 from queue.
803 315150 [MQTT] MQTT Unsubscribe was successful.
804 315150 [MQTT] Notifying task.
805 315151 [OTA Task] Command sent to MQTT task passed.
806 315152 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

807 315172 [OTA Task] Sending command to MQTT task.
808 315172 [MQTT] Received message e0000 from queue.
809 315273 [MQTT] MQTT Unsubscribe was successful.
810 315273 [MQTT] Notifying task.
811 315274 [OTA Task] Command sent to MQTT task passed.
812 315274 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

813 315275 [OTA Task] [OTA] Resetting MCU to activate new image.
0 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created

Device came up in Station mode

2 137 [Tmr Svc] Wi-Fi module initialized.
3 137 [Tmr Svc] Starting key provisioning...
4 137 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 339 [Tmr Svc] Write device certificate...
7 436 [Tmr Svc] Key provisioning done...
Device disconnected from the AP on an ERROR..!!

[WLAN EVENT] STA Connected to the AP: Guest , BSSID: 44:48:c1:ba:b2:c3

[NETAPP EVENT] IP acquired by the device


Device has connected to Guest

Device IP Address is 192.168.3.72


8 1443 [Tmr Svc] Wi-Fi connected to AP Guest.
9 1444 [Tmr Svc] IP Address acquired 192.168.3.72
10 1444 [OTA] OTA demo version 0.9.1
11 1445 [OTA] Creating MQTT Client...
12 1445 [OTA] Connecting to broker...
13 1445 [OTA] Sending command to MQTT task.
14 1445 [MQTT] Received message 10000 from queue.
15 2910 [MQTT] MQTT Connect was accepted. Connection established.
16 2910 [MQTT] Notifying task.
17 2911 [OTA] Command sent to MQTT task passed.
18 2912 [OTA] Connected to broker.
19 2913 [OTA Task] Sending command to MQTT task.
20 2913 [MQTT] Received message 20000 from queue.
21 3014 [MQTT] MQTT Subscribe was accepted. Subscribed.
22 3014 [MQTT] Notifying task.
23 3015 [OTA Task] Command sent to MQTT task passed.
24 3015 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/$next/get/
accepted

25 3028 [OTA Task] Sending command to MQTT task.
26 3028 [MQTT] Received message 30000 from queue.
27 3129 [MQTT] MQTT Subscribe was accepted. Subscribed.
28 3129 [MQTT] Notifying task.
29 3130 [OTA Task] Command sent to MQTT task passed.
30 3138 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/notify-next

31 3138 [OTA Task] [OTA] Check For Update #0
32 3138 [OTA Task] Sending command to MQTT task.
33 3138 [MQTT] Received message 40000 from queue.
```

```
34 3241 [MQTT] MQTT Publish was successful.
35 3241 [MQTT] Notifying task.
36 3243 [OTA Task] Command sent to MQTT task passed.
37 3245 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:TI-LaunchPad ]
38 3245 [OTA Task] [OTA] Set job doc parameter [ jobId:
 fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
39 3245 [OTA Task] [OTA] Identified job doc parameter [ self_test ]
40 3246 [OTA Task] [OTA] Set job doc parameter [ updatedBy: 589827 ]
41 3246 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
42 3246 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
43 3247 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
44 3247 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
45 3247 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
46 3247 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
47 3249 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa:
 Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
48 3249 [OTA Task] [OTA] Job is ready for self test.
49 3250 [OTA Task] Sending command to MQTT task.
51 3351 [MQTT] MQTT Publish was successful.
52 3352 [MQTT] Notifying task.
53 3352 [OTA Task] Command sent to MQTT task passed.
54 3353 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/jobs/
fe18c7ec_8c31_4438_b0b9_ad55acd95610/u55 3353 [OTA Task] Sending command to MQTT task.
56 3353 [MQTT] Received message 60000 from queue.
57 3455 [MQTT] MQTT Unsubscribe was successful.
58 3455 [MQTT] Notifying task.
59 3456 [OTA Task] Command sent to MQTT task passed.
60 3456 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

61 3456 [OTA Task] [OTA] Accepted final image. Commit.
62 3578 [OTA Task] Sending command to MQTT task.
63 3578 [MQTT] Received message 70000 from queue.
64 3779 [MQTT] MQTT Publish was successful.
65 3780 [MQTT] Notifying task.
66 3780 [OTA Task] Command sent to MQTT task passed.
67 3781 [OTA Task] [OTA] Published 'SUCCEEDED' status to $aws/things/TI-LaunchPad/jobs/
fe18c7ec_8c31_4438_b0b9_ad55acd95610/upd68 3781 [OTA Task] [OTA] Returned buffer to MQTT
 Client.
69 4251 [OTA] [OTA] Queued: 1   Processed: 1   Dropped: 0
70 4381 [OTA Task] [OTA] Missing job parameter: execution
71 4382 [OTA Task] [OTA] Missing job parameter: jobId
72 4382 [OTA Task] [OTA] Missing job parameter: jobDocument
73 4382 [OTA Task] [OTA] Missing job parameter: ts_ota
74 4382 [OTA Task] [OTA] Missing job parameter: files
75 4382 [OTA Task] [OTA] Missing job parameter: streamname
76 4382 [OTA Task] [OTA] Missing job parameter: certfile
77 4382 [OTA Task] [OTA] Missing job parameter: filepath
78 4383 [OTA Task] [OTA] Missing job parameter: filesize
79 4383 [OTA Task] [OTA] Missing job parameter: sig-sha1-rsa
80 4383 [OTA Task] [OTA] Missing job parameter: fileid
81 4383 [OTA Task] [OTA] Missing job parameter: attr
82 4383 [OTA Task] [OTA] Returned buffer to MQTT Client.
83 5251 [OTA] [OTA] Queued: 2   Processed: 2   Dropped: 0
```

Download, build, flash, and run the FreeRTOS OTA demo on the Texas Instruments CC3220SF-LAUNCHXL

To download FreeRTOS and the OTA demo code

1. Browse to the AWS IoT console and from the navigation pane, choose **Software**.

2. Under **FreeRTOS Device Software**, choose **Configure download**.
3. From the list of software configurations, choose **Connect to AWS IoT - TI**. Choose the configuration name, not the **Download** link.
4. Under **Libraries**, choose **Add another library**, and then choose **OTA Updates**.
5. Under **Demo Projects**, choose **OTA Updates**.
6. Under **Name required**, enter **`Connect-to-IoT-OTA-TI`**, and then choose **Create and download**.

Save the zip file that contains FreeRTOS and the OTA demo code to your computer.

To build the demo application

1. Extract the .zip file.
2. Follow the instructions in Getting Started with FreeRTOS (p. 71) 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.
3. Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_MQTT_DEMO_ENABLED`, and define `CONFIG_OTA_UPDATE_DEMO_ENABLED`.
4. Build the solution and make sure it builds without errors.
5. 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
6. 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 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created
Device came up in Station mode
2 142 [Tmr Svc] Wi-Fi module initialized.
3 142 [Tmr Svc] Starting key provisioning...
4 142 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 340 [Tmr Svc] Write device certificate...
7 433 [Tmr Svc] Key provisioning done...
[WLAN EVENT] STA Connected to the AP: Mobile , BSSID: 24:de:c6:5d:32:a4
[NETAPP EVENT] IP acquired by the device

Device has connected to Mobile
Device IP Address is 192.168.111.12

8 2666 [Tmr Svc] Wi-Fi connected to AP Mobile.
9 2666 [Tmr Svc] IP Address acquired 192.168.111.12
10 2667 [OTA] OTA demo version 0.9.2
11 2667 [OTA] Creating MQTT Client...
12 2667 [OTA] Connecting to broker...
13 3512 [OTA] Connected to broker.
14 3715 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/OtaGA/jobs/$next/
get/accepted
15 4018 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/OtaGA/jobs/notify-
next
```

```
16 4027 [OTA Task] [prvPAL_GetPlatformImageState] xFileInfo.Flags = 0250
17 4027 [OTA Task] [prvPAL_GetPlatformImageState] eOTA_PAL_ImageState_Valid
18 4034 [OTA Task] [OTA_CheckForUpdate] Request #0
19 4248 [OTA] [OTA_AgentInit] Ready.
20 4249 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken: 0:OtaGA ]
21 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
22 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
23 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
24 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: afr_ota
25 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
26 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: files
27 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
28 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
29 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
30 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
31 4251 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha1-rsa
32 4251 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
33 4251 [OTA Task] [prvOTA_Close] Context->0x2001b2c4
34 5248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
35 6248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
36 7248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
37 8248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
38 9248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
```

Download, build, flash, and run the FreeRTOS OTA demo on the Microchip Curiosity PIC32MZEF

To download the FreeRTOS OTA demo code

1. Browse to the AWS IoT console and from the navigation pane, choose **Software**.
2. Under **FreeRTOS Device Software**, choose **Configure download**.
3. From the list of software configurations, choose **Connect to AWS IoT - Microchip**. Choose the configuration name, not the **Download** link.
4. Under **Libraries**, choose **Add another library**, and then choose **OTA Updates**.
5. Under **Demo projects**, choose **OTA Update**.
6. Under **Name required**, enter a name for your custom FreeRTOS software configuration.
7. Choose **Create and download**.

To build the OTA update demo application

1. Extract the .zip file you just downloaded.
2. Follow the instructions in Getting Started with FreeRTOS (p. 71) 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.
3. Open `demos/include/aws_ota_codesigner_certificate.h`.
4. Paste the contents of your code-signing certificate into the `static const char signingcredentialSIGNING_CERTIFICATE_PEM` variable. Following 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";
```

5. Install Python 3 or later.
6. Install `pyOpenSSL` by running `pip install pyopenssl`.
7. 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`.
8. Open *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_MQTT_DEMO_ENABLED`, and define `CONFIG_OTA_UPDATE_DEMO_ENABLED`.
9. Build the solution and make sure it builds without errors.
10. 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
11. 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

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_MQTT_DEMO_ENABLED`, and define `CONFIG_OTA_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_MQTT_DEMO_ENABLED`, and define `CONFIG_OTA_UPDATE_DEMO_ENABLED`.
   - Make sure that your SHA-256/ECDSA code-signing certificate is copied into the `demos/include/aws_ota_codesigner_certificate.h`.

HTTPS demo applications

Overview

For examples using the FreeRTOS HTTPS Client library, see the HTTPS Client demo applications defined in `iot_demo_https_s3_download_async.c` and `iot_demo_https_s3_download_sync.c`.

The HTTPS Client demos show how to download a file from Amazon S3 using a pre-signed URL. The file is downloaded incrementally using HTTP Partial Content headers. The byte ranges for the size of the response body buffer are specified in each incremental request. The HTTPS Client library is a generic library that can be used to download files from other webservers as well. Please note that not all HTTP servers support a Partial Content download with a byte range.

To use the HTTPS Client demos, 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 needed for a TLS connection.

`keyCLIENT_PRIVATE_KEY_PEM`

The private key PEM needed for a TLS connection.

Configuration parameters

These configuration parameters apply to the HTTPS Client demo and need to be defined.

`IOT_DEMO_HTTPS_PRESIGNED_GET_URL`

The pre-signed URL for a GET request to Amazon S3 for a specific object. This must be of the form:

```
https://awsexamplebucket1.§3.amazonaws.com/object-key.txt?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1560555644&Signature=SomeHash12345UrlABcdEFgfIjK
%3D
```

Please see *freertos*`/demos/https/README.md` for instructions on generating a pre-signed URL using the Python script in the same folder.

These configuration parameters apply to the HTTPS Client demo and do not need to be defined, they are for more customization options.

`IOT_DEMO_HTTPS_PORT`

The HTTPS server TCP port to connect to. The default is 443.

`IOT_DEMO_HTTPS_TRUSTED_ROOT`

The trusted ROOT CA to connect to the HTTPS server. The HTTPS server is defined by the host name in the `IOT_DEMO_HTTPS_PRESIGNED_GET_URL`. The default is the Baltimore Cybertrust root certificate authority.

See HTTPS Client API Reference Demo Configuration for other configurations.

Usage instructions

Please see the HTTPS Client API Reference Demo Usage Instructions for more information and output examples.

AWS IoT Device Shadow demo application

FreeRTOS includes a demo application that demonstrates how to programmatically update and respond to changes in an AWS IoT Device Shadow. This demo application is defined in `FreeRTOS/demos/shadow/aws_iot_demo_shadow.c`. The device in this scenario is a light bulb whose color can be set to red or green.

The demo application creates three tasks:

- A main demo task that calls `prvShadowMainTask`.
- A device update task that calls `prvUpdateTask`.
- A number of shadow update tasks that call `prvShadowUpdateTasks`.

`prvShadowMainTask` initializes the Device Shadow client and initiates an MQTT connection to AWS IoT with the client credentials specified in `demos/include/aws_clientcredential.h`. The information specified in `aws_clientcredential.h`, including your your device's AWS IoT thing name and the MQTT broker endpoint and port, must be correct for the application to successfully connect to the AWS Cloud.

After the MQTT connection is established, the application creates the device update task. Finally, it creates shadow update tasks and then terminates. The `democonfigSHADOW_DEMO_NUM_TASKS` constant defined in `aws_iot_demo_shadow.c` controls the number of shadow update tasks created.

`prvShadowUpdateTasks` generates an initial thing shadow document and updates the shadow with the document. It then goes into an infinite loop that periodically updates the thing shadow's desired state, requesting that the light bulb change its color (from red to green to red).

`prvUpdateTask` responds to changes in the shadow's desired state. When the desired state changes, this task updates the reported state of the shadow to reflect the new desired state.

Before you can run the Device Shadow demo, you must complete the getting started First steps (p. 72) to set up AWS IoT and FreeRTOS so your device can communicate with the AWS Cloud.

After you set up AWS IoT and FreeRTOS, do the following:

1. Add the following policy to your device certificate:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:us-west-2:123456789012:client/yourClientId"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": "arn:aws:iot:us-west-2:123456789012:topicfilter/$aws/things/
thingName/shadow/*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource":
      "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/thingName/shadow/*"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource":
      "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/thingName/shadow/*"
    }
  ]
}
```

2. Open *freertos*`/vendors/`*vendor*`/boards/`*board*`/aws_demos/config_files/aws_demo_config.h`, comment out `#define CONFIG_MQTT_DEMO_ENABLED`, and define `CONFIG_SHADOW_DEMO_ENABLED`.
3. Build, flash, and run FreeRTOS to your device.

You can use the AWS IoT console to view your device's shadow and confirm that its desired and reported states change periodically.

1. In the AWS IoT console, from the left navigation pane, choose **Manage**.
2. Under **Manage**, choose **Things**, and then choose the thing whose shadow you want to view.
3. On the thing detail page, from the left navigation pane, choose **Shadow** to display the thing shadow.

For more information about how devices and shadows interact, see Device Shadow Service Data Flow.

Secure Sockets echo client demo

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. For more information about setting up a FreeRTOS-qualified board, see Getting Started with FreeRTOS.

To run the demo

> **Note**
> 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_MQTT_DEMO_ENABLED`, and define `CONFIG_TCP_ECHO_CLIENT_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`.

Using AWS IoT Device Tester for FreeRTOS

You can use AWS IoT Device Tester (IDT) for FreeRTOS to verify that the FreeRTOS operating system works locally on your device and can communicate with the AWS IoT Cloud. 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.

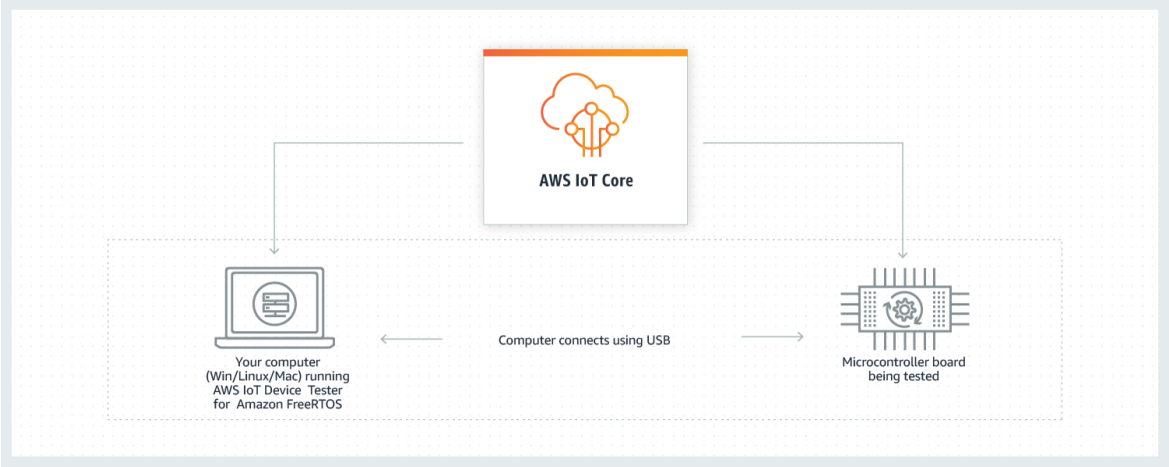
In addition to testing devices, IDT for FreeRTOS creates resources (for example, AWS IoT things, FreeRTOS groups, Lambda functions, and so on) to facilitate the qualification process.

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.

The following diagram shows the test infrastructure setup.



To run IDT for FreeRTOS, you can use the test resources. There are two types of resources:

- 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.

For more information, see AWS IoT Device Tester for FreeRTOS test suite versions (p. 292).

Supported versions of AWS IoT Device Tester for FreeRTOS

This topic lists supported versions of IDT 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. New releases of FreeRTOS might require you to download a new version of IDT for FreeRTOS.

By downloading the software, you agree to the IDT for FreeRTOS License Agreement.

Latest version of AWS IoT Device Tester for FreeRTOS

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

IDT v3.0.0 for FreeRTOS 202002.00

- IDT for FreeRTOS: Linux
- IDT for FreeRTOS: macOS
- IDT for FreeRTOS: Windows

  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.

Release notes

- Supports FreeRTOS 202002.00. For more information about what's included in the FreeRTOS 202002.00 release, 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 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 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 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.

Test suite versions

- FRQ_1.0.0

Earlier IDT versions for FreeRTOS

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

IDT v1.7.0 for FreeRTOS 202002.00

- IDT for FreeRTOS: Linux
- IDT for FreeRTOS: macOS
- IDT for FreeRTOS: Windows

Release notes

- Supports FreeRTOS 202002.00. For more information about what's included in the FreeRTOS 202002.00 release, 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 is misconfigured in the `device.json` file.
- Added an AWS Managed Policy `AWSIoTDeviceTesterForFreeRTOSFullAccess` with permissions required to run AWS IoT Device Tester. If new releases require additional permissions, we add them to this managed policy so that you don't have to manually update your IAM permissions.
- The file named `AFQ_Report.xml` in the results directory is now `FRQ_Report.xml`.

IDT v1.6.1 for FreeRTOS 201912.00

- IDT for FreeRTOS: Linux
- IDT for FreeRTOS: macOS
- IDT for FreeRTOS: Windows

Release notes

- Supports FreeRTOS 201912.00.
- 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.

IDT v1.5.2 for FreeRTOS 201910.00

- IDT for Amazon FreeRTOS: Linux
- IDT for Amazon FreeRTOS: macOS
- IDT for Amazon FreeRTOS: Windows

Release notes

- Supports qualification of FreeRTOS devices with secure element (onboard key).
- Supports configurable echo server ports for Secure Sockets and Wi-Fi test groups.
- Supports timeout multiplier flag to increase timeouts which comes in handy when you troubleshoot for timeout related errors.
- Added bug fix for log parsing.
- Supports iot ats endpoint in testing.

IDT v1.4.1 for FreeRTOS 201908.00

- IDT for Amazon FreeRTOS: Linux
- IDT for Amazon FreeRTOS: macOS
- IDT for Amazon FreeRTOS: Windows

Release notes

- Added support for new PKCS11 library and test case updates.
- Introduced actionable error codes. For more information, see IDT error codes (p. 296)
- Updated IAM policy used to run IDT.

For more information, see Support policy for AWS IoT Device Tester for FreeRTOS (p. 301).

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 Support policy for AWS IoT Device Tester for FreeRTOS (p. 301).

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

IDT v1.3.2 for FreeRTOS 201906.00 Major

- IDT for FreeRTOS: Linux
- IDT for FreeRTOS: macOS
- IDT for FreeRTOS: Windows

Release notes

- Added support for testing Bluetooth Low Energy (BLE).
- Improved user experience for IDT command line interface (CLI) commands.
- Updated IAM policy used to run IDT.

- **IDT-FreeRTOS v1.2**
  - FreeRTOS Versions:
    - FreeRTOS v1.4.9
    - FreeRTOS v1.4.8
  - Release Notes:
    - Added support for FreeRTOS v1.4.8 and v1.4.9.

    - Added support for testing FreeRTOS devices with the CMAKE build system.
- **IDT-FreeRTOS v1.1**
  - FreeRTOS Versions:
    - FreeRTOS v1.4.7
- **IDT-FreeRTOS v1.0**
  - FreeRTOS Versions:
    - FreeRTOS v1.4.6
    - FreeRTOS v1.4.5
    - FreeRTOS v1.4.4
    - FreeRTOS v1.4.3
    - FreeRTOS v1.4.2

Prerequisites

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

Download FreeRTOS

You can download the version of FreeRTOS that you want to test from GitHub. 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\`.

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 (p. 271).

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

Follow these steps to create and configure an AWS account, an IAM user, and an IAM policy that grants IDT for FreeRTOS permission to access resources on your behalf while running tests.

1. If you already have an AWS account, skip to the next step. Create an AWS account.
2. Create an IAM policy that grants IDT for FreeRTOS the IAM permissions to create service roles with specific permissions.

    a. Sign in to the IAM console.
    b. In the navigation pane, choose **Policies**.
    c. In the content pane, choose **Create policy**.
    d. Choose the **JSON** tab and copy the following permissions in to the **JSON** text box.

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 could potentially be used to gain administrator access to your AWS account.

Most Regions

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "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*"
            ]
        }
    ]
}
```

Beijing and Ningxia Regions

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

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreatePolicy",
                "iam:DetachRolePolicy",
                "iam:DeleteRolePolicy",
                "iam:DeletePolicy",
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:AttachRolePolicy"
            ],
            "Resource": [
                "arn:aws-cn:iam::*:policy/idt*",
                "arn:aws-cn:iam::*:role/idt*"
            ]
        }
    ]
}
```

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

f. On the **Review** page, enter `IDTFreeRTOSIAMPermissions` for the policy name. Review the policy **Summary** to verify the permissions granted by your policy.

g. Choose **Create policy**.

3. Create an IAM user with the necessary permissions to run AWS IoT Device Tester.

   a. Follow steps 1 through 5 in Creating IAM Users (Console).

   b. To attach the necessary permissions to your IAM user:

      i. On the **Set permissions** page, choose **Attach existing policies to user directly**.

      ii. Search for the **IDTFreeRTOSIAMPermissions** policy that you created in step 2. Select the check box.

      iii. Search for the **AWSIoTDeviceTesterForFreeRTOSFullAccess** policy. Select the check box.

   c. Choose **Next: Tags**.

   d. Choose **Next: Review** to view a summary of your choices.

   e. Choose **Create user**.

   f. To view the users' access keys (access key IDs and secret access keys), choose **Show** next to each password and access key and then choose **Download.csv**. Save the file to a safe location.

AWS IoT Device Tester managed policy

The `AWSIoTDeviceTesterForFreeRTOSFullAccess` managed policy contains the following permissions to enable device tester to execute and to collect metrics:

- `iot-device-tester:SupportedVersion`

  Grants permission to get the list of FreeRTOS versions and test suite versions supported by IDT, so that they are 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.

(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 in Install the AWS CLI.

Configure the 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.

Preparing to test your microcontroller board for the first time

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 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.

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 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 (p. 285).

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": "OTA",
      "value": "Yes | No",
          "configs": [
              {
                "name": "OTADataPlaneProtocol",
                "value": "HTTP | MQTT | Both"
              }
          ]
    },
```

```
    {
      "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"
    },
    {
      "name": "BLE",
      "value": "Yes | No"
    }
  ],
  "devices": [{
    "id": "device-id1",
    "connectivity": {
      "protocol": "uart",
      "serialPort": "computer_serial_port_1"
    },
    "secureElementConfig": {
        "publicKeyAsciiHexFilePath":"absolute-path-to-public-key",
    "secureElementSerialNumber": "optional: serialnumber-of-secure-element"
    },
    "identifiers": [{
      "name": "serialNo",
      "value": "serialNo-value"
    }]
},
{
    "id": "device-id2",
    "connectivity": {
      "protocol": "uart",
      "serialPort": "computer_serial_port_2"
    },
    "secureElementConfig": {
        "publicKeyAsciiHexFilePath": "absolute-path-to-public-key",
        "secureElementSerialNumber": "optional: serialnumber-of-secure-element"
    },
    "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. The 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.

`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. Currently, no board supports RSA only, so the choice `RSA` is invalid.

`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.

`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.

`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.

`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. The following JSON example shows how you can configure IDT for FreeRTOS for testing multiple devices:

```
{
  "sourcePath": "/absolute-path-to/freertos",
  "vendorPath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-name",
  "buildTool": {
    "name": "your-build-tool-name",
    "version": "your-build-tool-version",
    "command": [
      "/absolute-path-to/build-parallel.sh {{testData.sourcePath}} {{enableTests}}"
    ]
  },
  "flashTool": {
    "name": "your-flash-tool-name",
    "version": "your-flash-tool-version",
    "command": [
      "/absolute-path-to/flash-parallel.sh {{testData.sourcePath}}
 {{device.connectivity.serialPort}} {{buildImageName}}"
    ],
    "buildImageInfo" : {
      "testsImageName": "tests-image-name",
      "demosImageName": "demos-image-name"
    }
  },
  "clientWifiConfig": {
    "wifiSSID": "ssid",
    "wifiPassword": "password",
    "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA |
 eWiFiSecurityWPA2"
  },
  "testWifiConfig": {
    "wifiSSID": "ssid",
    "wifiPassword": "password",
    "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA |
 eWiFiSecurityWPA2"
  },
  "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"
  }
}
```

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

`sourcePath`

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

`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"
}
```

Note
When running tests in parallel, 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.

`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}}`.

- `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.

`buildTool.buildImageInfo.testsImageName`

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

`buildTool.buildImageInfo.demosImageName`

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

`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}}`.

`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.

- `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`

  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.

- `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`

  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.

`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 (p. 13).

`signerCertificateFileName`

The location of the code signing certificate on the device.

`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_certificate.h`.

`untrustedSignerCertificateArn`

The ARN for the code-signing certificate uploaded to ACM.

`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.

`otaDemoConfigFilePath`

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

`cmakeConfiguration`

CMake configuration [Optional]

`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

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.

`{{device.connectivity.serialPort}}`

Expands to the serial port.

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

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.

Running Bluetooth Low Energy tests

This section describes how to set up and run the Bluetooth 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 Preparing to test your microcontroller board for the first time (p. 277).
- A Raspberry Pi 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 3B+.

To set up your Raspberry Pi to run BLE tests

1. Download the custom Yocto image that contains the software required to perform the tests.
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-sdimg` 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 *`rasp-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-id-address"
                }
            }
        ]
    }
]
```

iii. (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-id-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"
        },
    ...
}
```

Running 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).

Troubleshooting BLE tests

Make sure you have followed the steps in Preparing to test your microcontroller board for the first time (p. 277). If tests other than BLE are failing, then the problem is most likely not due to the Bluetooth configuration.

Running 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  \
    --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_1.0.0  \
    --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.

Note
Starting in IDT v3.0.0, IDT checks online for newer test suites. For more information, see Test suite versions (p. 292).

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
```

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

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

```
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 Running the FreeRTOS qualification suite (p. 289).

Note
Starting in IDT v3.0.0, IDT checks online for newer test suites. For more information, see Test suite versions (p. 292).

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.

AWS IoT Device Tester for FreeRTOS test suite versions

IDT for FreeRTOS organizes tests 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 TBD 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 Support policy for AWS IoT Device Tester for FreeRTOS (p. 301).

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.

Understanding results and logs

This section describes how to view and interpret IDT result reports and logs.

Viewing 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).

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.

Interpreting 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 <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 Troubleshooting (p. 295).

Viewing 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.

Troubleshooting

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 (p. 274).
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 logs 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.
   - Make sure that you have a clean, cloned version of the FreeRTOS source code. FreeRTOS releases are tagged according to FreeRTOS version. To clone a specific version of the code, use git clone --branch *version-number* https://github.com/aws/amazon-freertos.git.

Troubleshooting 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`, or `userdata.json` are either missing or in an incorrect format.	Make sure required fields are not missing and are in required format in listed files. For more information, see Preparing to test your microcontroller board for the first time (p. 277).
202	ValidationError	Fields in `device.json`, `config.json`, 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.json` 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 configured correctly. For more information, see Create and configure an AWS account (p. 274).

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 (p. 280).
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 (p. 280).
205	FlashOrRunTestError	IDT FreeRTOS is unable to flash or run FreeRTOS on your DUT.	Verify the information under `flashTool` in your `userdata.json` file is correct. For more information, see Configure build, flash, and test settings (p. 280).
206	StartEchoServerError	IDT FreeRTOS is unable to start echo server for the WiFi or secure sockets tests.	Verify the ports configured under `echoServerConfiguration` in your `userdata.json` file are not in use or blocked by firewall or network settings.

Debugging 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.

Debugging a "required parameter missing" error

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.

Debugging a "could not start test" error

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.

Debugging 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 (p. 274).

Debugging 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 (p. 280). 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."

Debugging 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.

Troubleshooting 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_1.0.0 --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
```

Support policy for AWS IoT Device Tester for FreeRTOS

AWS IoT Device Tester for FreeRTOS is a test automation tool to validate and qualify your FreeRTOS devices for inclusion in the AWS Partner Device Catalog. We recommend that you use the most recent version of FreeRTOS and AWS IoT Device Tester to test or qualify your devices. We support AWS IoT Device Tester for the most recent version of FreeRTOS and for FreeRTOS versions released within the previous six months. The latest version of FreeRTOS is available on GitHub. For supported versions of AWS IoT Device Tester, see Supported versions of AWS IoT Device Tester for FreeRTOS (p. 271).

For each version of the IDT framework, three versions of the test suite will be supported for qualification of devices.

You can also use any of the supported versions of AWS IoT Device Tester with the corresponding version of FreeRTOS to test or qualify your devices. Although you can continue to use Unsupported IDT versions for FreeRTOS (p. 273), these will not receive bug fixes or updates.

If you have 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 AWS resources (p. 302)
- Compliance validation (p. 313)
- Resilience in AWS (p. 313)
- Infrastructure security in FreeRTOS (p. 313)

Identity and Access Management for AWS resources

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 AWS resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience (p. 303)
- Authenticating with identities (p. 303)
- Managing access using policies (p. 305)
- Learn more (p. 306)
- How AWS services work with IAM (p. 306)
- Identity-based policy examples (p. 309)
- Troubleshooting identity and access (p. 311)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work you do in AWS.

Service user – If you use AWS services to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more 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 an AWS service, see Troubleshooting identity and access (p. 311).

Service administrator – If you're in charge of AWS resources at your company, you probably have full access to the services you use. It's your job to determine which features and resources your employees 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 AWS, see How AWS services work with IAM (p. 306).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS. For more information about AWS identity-based policies that you can use in IAM, see Policies and Permissions in the AWS Identity and Access Management User Guide.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see The IAM Console and Sign-in Page in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication, or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email or your IAM user name. You can access AWS programmatically using your root user or IAM user access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also 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 Using Multi-Factor Authentication (MFA) in AWS in the *IAM User Guide*.

AWS Account Root User

When you first create an AWS account, you begin with a single 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 do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing Access Keys for IAM Users in the *IAM*

User Guide. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

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 When to Create an IAM User (Instead of a Role) 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. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling a AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM Roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the *IAM User Guide*.
- **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 How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.
- **AWS service access** – A service role is an IAM role that a service assumes to perform actions in your account on your behalf. When you set up some AWS service environments, you must define a role for the service to assume. This service role must include all the permissions that are required for the service to access the AWS resources that it needs. Service roles vary from service to service, but many allow you to choose your permissions as long as you meet the documented requirements for that service. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. You can create, modify, and delete a service role from within IAM. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.
- **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 Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances in the *IAM User Guide*.

To learn whether to use IAM roles, see When to Create an IAM Role (Instead of a User) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when an entity (root user, IAM user, or IAM role) 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*.

An IAM administrator can use policies to specify who has access to AWS resources, and what actions they can perform on those resources. Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

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, role, or group. These policies control what actions that identity can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM 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 Choosing 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 such as an Amazon S3 bucket. Service administrators can use these policies to define what actions a specified principal (account member, user, or role) can perform on that resource and under what conditions. Resource-based policies are inline policies. There are no managed resource-based policies.

Access Control Lists (ACLs)

Access control policies (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they are the only policy type that does 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 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 How SCPs Work 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*.

Learn more

For more information about identity and access management for AWS resources, continue to the following pages:

- How AWS services work with IAM (p. 306)
- Identity-based policy examples (p. 309)
- Troubleshooting identity and access (p. 311)

How AWS services work with IAM

Before you use IAM to manage access to AWS services, you should understand what IAM features are available to use. To get a high-level view of how AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

- Identity-based policies (p. 306)
- AWS resource-based policies (p. 308)
- Authorization based on tags (p. 308)
- IAM roles (p. 308)

Identity-based policies

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. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

The `Action` element of an IAM identity-based policy describes the specific action or actions that will be allowed or denied by the policy. Policy actions usually have the same name as the associated AWS API operation. The action is used in a policy to grant permissions to perform the associated operation.

Policy actions use a prefix before the action. Policy statements must include either an `Action` or `NotAction` element. Each service defines its own set of actions that describe tasks that you can perform with the service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "service-prefix:action1",
      "service-prefix:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action:

```
"Action": "service-prefix:Describe*"
```

To see a list of AWS actions, see Actions, Resources, and Condition Keys for AWS Services in the *IAM User Guide*.

Resources

The `Resource` element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. You specify a resource using an ARN or using the wildcard (*) to indicate that the statement applies to all resources.

For more information about the format of ARNs, see Amazon Resource Names (ARNs) and AWS Service Namespaces.

To specify all instances that belong to a specific account, use the wildcard (*):

```
"Resource": "arn:aws:service-prefix:us-east-1:123456789012:resource-type/*"
```

Some actions, such as those for creating resources, cannot be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

Some API actions involve multiple resources, so an IAM user must have permissions to use all the resources. To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
      "resource1",
      "resource2"
```

To learn with which actions you can specify the ARN of each resource, see Actions, Resources, and Condition Keys for AWS Services.

Condition keys

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can build 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*.

To see all AWS global condition keys, see AWS Global Condition Context Keys in the *IAM User Guide*."

AWS resource-based policies

Resource-based policies are JSON policy documents that specify what actions a specified principal can perform on a resource and under what conditions. Resource-based policies let you grant usage permission to other accounts on a per-resource basis.

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, you must also grant the principal entity permission to access the resource. 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 How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.

To view an example of a detailed resource-based policy page, see https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html.

Authorization based on tags

You can attach tags to resources or pass tags in a request. To control access based on tags, you provide tag information in the condition element of a policy using the *`prefix`*`:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Viewing resources based on tags (p. 310).

IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS Security Token Service (AWS STS) API operations such as AssumeRole or GetFederationToken.

Service-linked roles

Service-linked roles allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view but not edit the permissions for service-linked roles.

Service roles

This feature allows a service to assume a service role on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

Identity-based policy examples

By default, IAM users and roles don't have permission to create or modify AWS resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

Topics

- Policy best practices (p. 309)
- Using the AWS console (p. 309)
- Allow users to view their own permissions (p. 310)
- Viewing resources based on tags (p. 310)

Policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete 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 Using AWS Managed Policies** – To start using AWS services quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get Started Using Permissions With AWS Managed Policies in the *IAM User Guide*.
- **Grant Least Privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant Least Privilege in the *IAM User Guide*.
- **Enable MFA for Sensitive Operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using Multi-Factor Authentication (MFA) in AWS in the *IAM User Guide*.
- **Use Policy Conditions for Extra Security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON Policy Elements: Condition in the *IAM User Guide*.

Using the AWS console

To access an AWS service console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the 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 (IAM users or roles) with that policy.

To ensure that those entities can still use the console, also attach an AWS managed policy to the entities. For more information, see Adding Permissions to a User in the *IAM User Guide*.

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 you're trying to perform.

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": "*"
           }
       ]
   }
```

Viewing resources based on tags

You can use conditions in your identity-based policy to control access to resources based on tags. This example shows how you might create a policy that allows viewing a resource. However, permission is granted only if the resource tag `Owner` has the value of that user's user name. This policy also grants the permissions necessary to complete this action on the console.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListInputsInConsole",
            "Effect": "Allow",
```

```
            "Action": "prefix:ListInputs",
            "Resource": "*"
        },
        {
            "Sid": "ViewResourceIfOwner",
            "Effect": "Allow",
            "Action": "prefix:ListInputs",
            "Resource": "arn:aws:prefix:*:*:resource-name/*",
            "Condition": {
                "StringEquals": {"prefix:ResourceTag/Owner": "${aws:username}"}
            }
        }
    ]
}
```

You can attach this policy to the IAM users in your account. If a user named `richard-roe` attempts to view a *`resource-name`*, the *`resource-name`* must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise he is denied access. The condition tag key `Owner` matches both `Owner` and `owner` because condition key names are not case-sensitive. For more information, see IAM JSON Policy Elements: Condition in the *IAM User Guide*.

Troubleshooting identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with IAM.

Topics

- I am not authorized to perform an action (p. 311)
- I am not authorized to perform iam:PassRole (p. 311)
- I want to view my access keys (p. 312)
- I'm an administrator and want to allow others to access AWS resources (p. 312)
- I want to allow people outside of my AWS account to access my resources (p. 312)

I am not authorized to perform an action

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a *`my-example-resource`* but does not have *`prefix`*`:Action` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to
 perform: prefix:Action on resource: my-example-resource
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`my-example-resource`* using the *`prefix`*`:Action`.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to a service.

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 a service. However, the action requires the service to have permissions 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 asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

> **Important**
> Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing Access Keys in the *IAM User Guide*.

I'm an administrator and want to allow others to access AWS resources

To allow others to access a service, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in AWS.

To get started right away, see Creating Your First IAM Delegated User and Group in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my 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 a service supports these features, see How AWS services work with IAM (p. 306).
- 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 How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.

Compliance validation

FreeRTOS is not in scope of any AWS compliance programs. For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. 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 FreeRTOS 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 and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Config – This AWS 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 that helps you check your compliance with security industry standards and best practices.

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) 1.0 or later. We recommend TLS 1.2 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.