aws

ユーザーガイド

FreeRTOS



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

FreeRTOS: ユーザーガイド

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

Amazon の商標およびトレードドレスはAmazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

FreeRTOS とは 1
FreeRTOS ソースコードをダウンロードする 1
FreeRTOS 認定ハードウェアプラットフォーム 1
追加リソース 2
FreeRTOS バージョン 3
FreeRTOS 長期サポート 3
FreeRTOS 拡張メンテナンスプラン 4
FreeRTOS アーキテクチャ 4
開発ワークフロー 5
FreeRTOS カーネルの基礎 6
FreeRTOS カーネルスケジューラ 6
カーネルメモリの割り当て 7
アプリケーションメモリを管理する 7
タスク間の調整 8
キュー 8
セマフォとミューテックス 9
直接タスク通知 9
ストリームバッファ 10
メッセージバッファ 11
対称型マルチプロセッシング (SMP) のサポート 13
ソフトウェアタイマー 14
低電力サポート 14
FreeRTOSConfig.h 14
AWS IoT 埋め込み C 用の Device SDK 16
FreeRTOS 共通 IO APIs 17
[Libraries] (ライブラリ) 17
共通 IO - ベーシック 17
共通 IO - BLE 19
Amazon 共通ソフトウェア用の共通 IO 19
ACS とは 19
認定プログラム 20
FreeRTOS の使用を開始する 21
Quick Connect の使用を開始する 21
FreeRTOS ライブラリの詳細を理解する 21

安全で堅牢な AWS IoT 製品を構築する 21
AWS IoT アプリケーションを開発する 22
AWS IoT Device Tester FreeRTOS 用 23
FreeRTOS 認定スイート 23
カスタムテストスイートを理解する 24
IDT for FreeRTOS のサポートされているバージョン 25
IDT for FreeRTOS の最新バージョン 25
IDT の以前のバージョン 27
IDT のサポートされていないバージョン 33
IDT for FreeRTOS のダウンロード 63
IDT を手動でダウンロードする 63
IDT をプログラムでダウンロード 64
IDT with FreeRTOS 認定スイート 2.0 (FRQ 2.0) 69
LTS 認定の前提条件を設定する 70
マイクロコントローラーボードの最初のテスト 80
IDT UI を使用して FreeRTOS 認定スイートを実行する 97
FreeRTOS 認定 2.0 スイートを実行する 112
IDT for FreeRTOSresultsを表示する 116
IDT for FreeRTOS の結果の解釈 116
IDT for FreeRTOSlogs を表示する 119
IDT with FreeRTOS 認定スイート 1.0 (FRQ 1.0) 119
1.0 認定の前提条件を設定する 121
マイクロコントローラーボードの最初のテスト 125
IDT UI を使用して FreeRTOS 認定スイートを実行する 145
Bluetooth Low Energy テストを実行する 156
FreeRTOS 認定スイートを実行する 162
IDT for FreeRTOS の結果を表示する 167
IDT for FreeRTOS の結果の解釈 168
IDT for FreeRTOS ログを表示する 171
独自の IDT テストスイートを開発して実行する 172
最新バージョンの IDT for FreeRTOS をダウンロードする 172
テストスイートワークフロー 173
チュートリアル: サンプル IDT テストスイートを構築して実行する 173
チュートリアル: シンプルな IDT テストスイートの開発 179
テストスイートのバージョン 266
エラーのトラブルシューティング 267

デバイス設定のトラブルシューティング 268
タイムアウトエラーのトラブルシューティング 281
セルラー機能と AWS 料金 282
認定レポート生成ポリシー 282
AWS の管理ポリシー AWS IoT Device Tester 282
マネージドポリシー 283
ポリシーの更新 290
サポートポリシー 292
のセキュリティ AWS 294
Identity and Access Management 294
対象者 295
アイデンティティを使用した認証 296
ポリシーを使用したアクセスの管理 299
FreeRTOS と IAM の連携の仕組み 302
アイデンティティベースのポリシーの例 309
トラブルシューティング 312
コンプライアンス検証 314
耐障害性 315
インフラストラクチャセキュリティ 315
Amazon FreeRTOS Github リポジトリ移行ガイド 317
付録 317
アーカイブ 324
FreeRTOS ユーザーガイドのアーカイブ 324
以前の「FreeRTOS ユーザーガイド」のコンテンツ 324
FreeRTOS の使用を開始する 324
無線による更新 529
FreeRTOS ライブラリ 613
FreeRTOS デモ 681
..... dcccviii

FreeRTOS とは

世界をリードするチップ企業との 15 年間にわたる提携によって開発され、現在 170 秒ごとにダウンロードされている FreeRTOS は、マイクロコントローラーおよび小型マイクロプロセッサ向けの市場をリードするリアルタイムオペレーティングシステム (RTOS) です。MIT オープンソースライセンスで無料配布されている FreeRTOS には、すべての業種での使用に適したカーネルと増え続けるライブラリのセットが含まれています。FreeRTOS は、信頼性と使いやすさを重視して構築されています。

FreeRTOS には、接続性、セキュリティ、無線通信経由 (OTA) の更新を実現するライブラリが備わっています。さらに FreeRTOS には、認定済みボードに FreeRTOS の機能を表示するデモアプリケーションも含まれています。

FreeRTOS はオープンソースプロジェクトです。ソースコードをダウンロードしたり、変更や機能強化を提供したり、https://github.com/FreeRTOS/FreeRTOS の GitHub サイトで問題を報告したりできます。

MIT オープンソースライセンスに基づいて FreeRTOS コードをリリースしているため、商用および個人用のプロジェクトで使用できます。

また、FreeRTOS ドキュメント (FreeRTOS ユーザーガイド、FreeRTOS 移植ガイド、FreeRTOS 資格ガイド) への投稿も歓迎します。ドキュメントのマークダウンソースを確認する場合は、https://github.com/awsdocs/aws-freertos-docs を参照してください。これは、クリエイティブコモンズ (CC BY-ND) ライセンスに基づいてリリースされています。

FreeRTOS ソースコードをダウンロードする

freertos.org のダウンロードページから最新の FreeRTOS と長期サポート (LTS) パッケージをダウンロードします。

FreeRTOS 認定ハードウェアプラットフォーム

以下のハードウェアプラットフォームは、FreeRTOS に認定されています。

- 用 ATECC608A ゼロタッチプロビジョニングキット AWS IoT
- Cypress CYW943907AEVAL1F 開発キット

- Cypress CYW954907AEVAL1F 開発キット
- Cypress CY8CKIT-064S0S2-4343W キット
- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- Espressif ESP-WROOM-32SE
- Espressif ESP32-S2-Saola-1
- Infineon XMC4800 IoT 接続キット
- Marvell MW320 AWS IoT Starter キット
- Marvell MW322 AWS IoT Starter Kit
- MediaTek MT7697Hx Development Kit
- Microchip Curiosity PIC32MZEF バンドル
- Nordic nRF52840-DK
- NuMaker-IoT-M487
- NXP LPC54018 IoT モジュール
- OPTIGA Trust X セキュリティソリューション
- Renesas RX65N RSK IoT モジュール
- STMicroelectronicsSTM32L4 Discovery Kit IoT Node
- Texas Instruments CC3220SF-LAUNCHXL
- Microsoft Windows 7 以降 (最低でもデュアルコアで有線イーサネット接続があること)
- Xilinx Avnet MicroZed Industrial IoT キット

認定済みデバイスの一覧については、AWS パートナーデバイスカタログを参照してください。

新しいデバイスの資格認定については、FreeRTOS 資格ガイドを参照してください。

追加リソース

これらのリソースが役に立つ場合があります。

- その他の FreeRTOS ドキュメントについては、freertos.org を参照してください。
- FreeRTOS について FreeRTOS エンジニアリングチームに質問がある場合は、FreeRTOS の GitHub ページで問題を提出してください。

- FreeRTOS に関する技術的な質問については、FreeRTOS コミュニティフォーラムを参照してください。
- デバイスの への接続の詳細については AWS IoT、「 AWS IoT Core デベロッパーガイド」の「デバイスプロビジョニング」を参照してください。
- のテクニカルサポートについては AWS、AWS 「 サポート」を参照してください。
- の AWS 請求、アカウントサービス、イベント、不正使用、またはその他の問題については AWS、お問い合わせページを参照してください。

FreeRTOS バージョン

個々のライブラリは、セマンティックバージョニングと同様の x.y.z スタイルのバージョン番号を使用します。x はメジャーバージョン番号、y はマイナーバージョン番号、そして 2022 年以降の z はパッチ番号です。2022 年より前は、z はポイントリリース番号でした。そのため、最初の LTS ライブラリには x.y.z LTS Patch 2 という形式のパッチ番号が必要でした。

ライブラリパッケージは yyyymm.x スタイルの日付スタンプバージョン番号を使用します。yyyy は年、mm は月、x は月内のリリース順序を示すオプションのシーケンス番号です。LTS パッケージの場合、x はその LTS リリースのシーケンスパッチ番号です。パッケージに含まれる個々のライブラリは、その日付における最新バージョンのライブラリです。LTS パッケージの場合、その日に LTS バージョンとして最初にリリースされた LTS ライブラリの最新のパッチバージョンです。

FreeRTOS 長期サポート

FreeRTOS 長期サポート (LTS) リリースは、リリースから少なくとも 2 年間、セキュリティと重大なバグ修正 (必要な場合) を受けます。この継続的なメンテナンスにより、FreeRTOS ライブラリの新しいメジャーバージョンへの更新で費用のかかるダウンタイムを発生させることなく、開発とデプロイのサイクル全体にわたってバグ修正を組み込むことができます。

FreeRTOS LTS を使用すると、セキュアなコネクテッド IoT および組み込み製品の構築に必要なライブラリの完全なセットを入手できます。LTS は、既に実稼働環境にあるデバイス上のライブラリの更新に関連するメンテナンスおよびテストのコストを削減するのに役立ちます。

FreeRTOS LTS にはFreeRTOS カーネルと IoT ライブラリ、FreeRTOS +TCP、coreMQTT、coreHTTP、corePKCS11、coreJSON、 AWS IoT OTA、 AWS IoT Jobs AWS IoT Device Defender、 AWS IoT Device Shadow が含まれています。詳細については、FreeRTOS の LTS ライブラリを参照してください。

FreeRTOS 拡張メンテナンスプラン

AWS はFreeRTOS 延長メンテナンスプラン (EMP) も提供します。これにより、選択した FreeRTOS 長期サポート (LTS) バージョンでセキュリティパッチと重要なバグ修正を最大 10 年間追加できます。FreeRTOS EMP を使用すると、FreeRTOS ベースの長寿命デバイスでは、機能の安定性を備え、セキュリティアップデートを長期間受け取るバージョンを利用できます。FreeRTOS ライブラリの今後のパッチの通知が適時届くので、IoT デバイスへのセキュリティパッチのデプロイを計画できます。

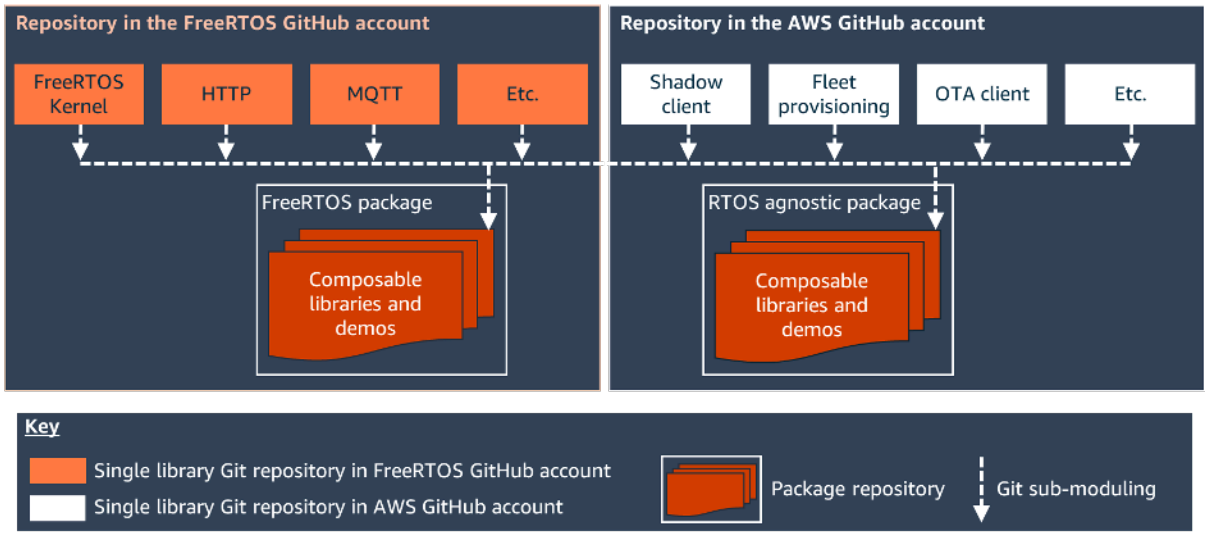
FreeRTOS EMP の詳細については、機能ページを参照してください。

FreeRTOS アーキテクチャ

FreeRTOS には、単一ライブラリリポジトリとパッケージリポジトリの 2 種類のリポジトリが含まれています。各単一ライブラリリポジトリには、ビルドプロジェクトやサンプルがない 1 つのライブラリのソースコードが含まれています。パッケージリポジトリには複数のライブラリが含まれており、ライブラリの使用方法を示す事前構成済みプロジェクトを含めることができます。

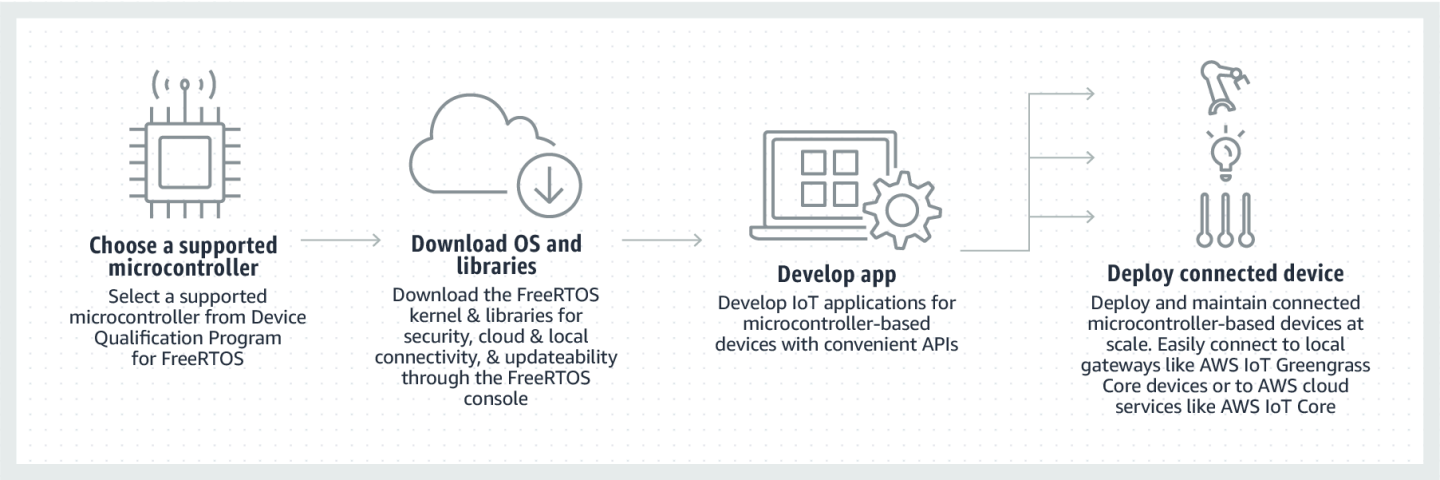
パッケージリポジトリには複数のライブラリがありますが、ライブラリのコピーが含まれているわけではありません。代わりに、パッケージリポジトリに含まれるライブラリを git サブモジュールとして参照します。サブモジュールを使用すると、個々のライブラリの信頼できる唯一のソースが確保できます。

個々のライブラリの git リポジトリは 2 つの GitHub 組織に分割されます。FreeRTOS 固有のライブラリ (FreeRTOS+TCP など) または汎用ライブラリ (任意の MQTT ブローカーで動作するため、クラウド非依存型である coreMQTT など) を含むリポジトリは、FreeRTOS GitHub 組織にあります。AWS IoT 特定のライブラリ (AWS IoT over-the-air更新クライアントなど) を含むリポジトリは、AWS GitHub 組織にあります。次の図は、その構成を示しています。



開発ワークフロー

開発を開始するには、FreeRTOS をダウンロードします。パッケージを解凍し、IDE にインポートします。その後、選択したハードウェアプラットフォームでアプリケーションを開発し、デバイスに適した開発プロセスを使用してこれらのデバイスを製造およびデプロイすることができます。デプロイされたデバイスは、 AWS IoT サービスに接続することも、完全な IoT ソリューション AWS IoT Greengrass の一部として接続することもできます。



FreeRTOS カーネルの基礎

FreeRTOS カーネルは、多くのアーキテクチャをサポートするリアルタイムのオペレーティングシステムです。その基本は、組み込みマイクロコントローラーアプリケーションの構築に最適です。次の機能があります。

- マルチタスクスケジューラ。
- 複数のメモリ割り当てオプション (完全に静的に割り当てられたシステムを作成する機能を含む)。
- タスク通知、メッセージキュー、複数タイプのセマフォ、ストリームおよびメッセージバッファを含むタスク間調整のプリミティブ。
- マルチコアマイクロコントローラーでの対称型マルチプロセッシング (SMP) のサポート。

FreeRTOS カーネルは、クリティカルなセクションや割り込みの中でリンクされたリストを処理するなど、非決定的なオペレーションは実行しません。FreeRTOS カーネルには、タイマーが処理を必要としない限り CPU 時間を使用しない効率的なソフトウェアタイマーが実装されています。ブロックされたタスクは、時間を消費する定期的な処理を必要としません。タスクへのダイレクト通知により、実質的に RAM オーバーヘッドが無くなり、タスクシグナリングが高速になります。ほとんどのタスク間およびinterrupt-to-taskシグナリングシナリオで使用できます。

FreeRTOS カーネルは、小さく、シンプルで使いやすく設計されています。一般的な RTOS カーネルバイナリイメージは、4000〜9000 バイトの範囲です。

FreeRTOS カーネルに関する最新のドキュメントについては、FreeRTOS.org を参照してください。FreeRTOS.org には、FreeRTOS カーネルの使用に関する詳細なチュートリアルとガイドが多数用意されています。これには、FreeRTOS FreeRTOS カーネルクイックスタートガイドやFreeRTOS ドキュメントのより詳細な RTOS 実装が含まれます。

FreeRTOS カーネルスケジューラ

RTOS を使用する組み込みアプリケーションは、独立したタスクのセットとして構成できます。各タスクは、他のタスクに依存することなく、独自のコンテキスト内で実行されます。どの時点においても、アプリケーション内の 1 つのタスクだけが実行されます。リアルタイム RTOS スケジューラは、各タスクの実行時期を決定します。各タスクには独自のスタックが用意されています。他のタスクを実行できるようにタスクをスワップすると、そのタスクの実行コンテキストがタスクスタックに保存されるため、後で同じタスクをスワップバックして実行を再開すると復元できます。

決定的なリアルタイム動作を提供するために、FreeRTOS タスクスケジューラは、タスクに厳密な優先順位を割り当てます。RTOS は、実行可能な最優先タスクに確実に処理時間が与えられるようにします。これは、同時に実行する準備ができている場合に、等しい優先順位のタスク間で処理時間を共有することを意味します。FreeRTOS は、実行準備が整っているタスクが他にない場合にのみ実行されるアイドルタスクも作成します。

カーネルメモリの割り当て

RTOS カーネルは、タスク、キュー、または他の RTOS オブジェクトが作成されるたびに RAM を必要とします。RAM は以下のように割り当てることができます。

- コンパイル時に静的に
- RTOS API オブジェクト作成関数によって RTOS ヒープから動的に

RTOS オブジェクトを動的に作成する場合、標準の C ライブラリ `malloc()` と `free()` 関数を使用することは、次の理由から必ずしも適切ではありません。

- 組み込みシステムでは使用できない可能性があります。
- 貴重なコードスペースを占有します。
- 通常はスレッドセーフではありません。
- 決定的ではありません。

これらの理由から、FreeRTOS はメモリ割り当て API をポータブル層に保持します。ポータブル層は、コア RTOS 機能を実装するソースファイルの外部にあるため、開発中のリアルタイムシステムに適したアプリケーション固有の実装を提供できます。RTOS カーネルに RAM が必要な場合は、`malloc()` の代わりに `pvPortMalloc()` を呼び出します。RAM が解放されると、RTOS カーネルは `free()` の代わりに `vPortFree()` を呼び出します。

アプリケーションメモリを管理する

アプリケーションがメモリを必要とする場合、FreeRTOS ヒープからメモリを割り当てることができます。FreeRTOS には、複雑さと機能に幅があるいくつかのヒープ管理スキームがあります。独自のヒープ実装を提供することもできます。

FreeRTOS カーネルには、次の 5 つのヒープ実装が含まれています。

heap_1

最も簡単な実装です。メモリを解放することはできません。

heap_2

メモリを解放することはできますが、フリーブロックに隣接するメモリを結合することはできません。

heap_3

スレッドの安全性のために標準の `malloc()` と `free()` をラップします。

heap_4

断片化を避けるために、隣接するフリーブロックを結合します。絶対アドレス配置オプションを含みます。

heap_5

これは heap_4 に似ています。ヒープは複数の隣接していないメモリ領域にまたがることができます。

タスク間の調整

このセクションには、FreeRTOS プリミティブについての情報が含まれています。

トピック

- キュー
- セマフォとミューテックス
- 直接タスク通知
- ストリームバッファ
- メッセージバッファ

キュー

キューは、タスク間通信の主要な形式です。タスク間や、割り込みとタスク間でメッセージを送信するために使用できます。ほとんどの場合、スレッドセーフな先入れ先出し (FIFO) バッファとして使用され、新しいデータがキューの後ろに送られます。(データはキューの先頭に送ることもできます)

メッセージはコピーでキューに送られます。つまり、データへの参照を単に格納するのではなく、データ (より大きなバッファへのポインタでも可能) 自体がキューにコピーされます。

キュー API は、ブロック時間を指定することを許可します。タスクが空のキューからの読み取りを試行すると、タスクは、データがキューで使用可能になるか、ブロック時間が経過するまでブロック状態になります。ブロック状態のタスクは CPU 時間を消費せずに他のタスクを実行できます。同様に、タスクがフルキューに書き込もうとすると、タスクはキュー内のスペースが使用可能になるかブロック時間が経過するまでブロック状態になります。複数のタスクが同じキューでブロックされている場合は、優先度の最も高いタスクが最初にブロック解除されます。

タスクへのダイレクト通知やストリーム、メッセージバッファなどの他の FreeRTOS プリミティブは、多くの一般的な設計シナリオでキューに対する軽量の代替手段を提供します。

セマフォとミューテックス

FreeRTOS カーネルは、相互排除と同期のためにバイナリセマフォ、カウントセマフォ、およびミューテックスを提供します。

バイナリセマフォは 2 つの値しか持てません。これらは、(タスク間またはタスクと割り込みの間の) 同期の実装に適しています。カウンティングセマフォは、2 つ以上の値を持てます。これにより、多くのタスクがリソースを共有したり、より複雑な同期操作を実行できます。

ミューテックスは、優先度継承メカニズムを含むバイナリセマフォです。つまり、現在優先度の低いタスクが保持しているミューテックスを取得しようとしている際に優先度の高いタスクがブロックした場合、トークンを保持しているタスクの優先度を一時的にブロックタスクの優先度に上げます。このメカニズムは、発生した優先度逆転を最小限に抑えるために、より高い優先度のタスクのブロックされた状態ができるだけ短時間になるよう設計されています。

直接タスク通知

タスク通知により、セマフォのような別個の通信オブジェクトを必要とせずに、タスクは他のタスクとやり取りし、割り込みサービスルーチン (ISR) と同期することができます。各 RTOS タスクには、通知に内容があればそれを格納するために使用される 32 ビットの通知値があります。RTOS タスク通知は、受信タスクのブロックを解除し、オプションで受信タスクの通知値を更新することができるタスクに直接送信されるイベントです。

RTOS タスク通知は、バイナリとカウンティングセマフォ、場合によってはキューの代わりに、より高速で軽量の代替として使用できます。タスク通知は、同等の機能を実行できる他の FreeRTOS 機能よりも、スピードおよび RAM フットプリントの両方で利点があります。ただし、タスク通知は、イベントの受信側になることができるタスクが 1 つしかない場合にのみ使用できます。

ストリームバッファ

ストリームバッファは、バイトのストリームを割り込みサービスルーチンからタスクに、またはあるタスクから別のタスクに渡すことができます。バイトストリームは任意の長さにすることができ、必ずしも先頭または末尾を必要としません。任意の数のバイトを一度に書き込み、および読み取りすることができます。プロジェクトに stream_buffer.c ソースファイルを含めることで、ストリームバッファ機能を有効にします。

ストリームバッファは、バッファ (ライター) に書き込むタスクまたは割り込みが 1 つだけであり、バッファ (リーダー) から読み取るタスクまたは割り込みが 1 つしかないことを前提としています。ライターとリーダーが異なるタスクになることやサービスルーチンを中断することは安全と言えますが、複数のライターやリーダーがあるのは安全とは言えません。

ストリームバッファの実装では、タスクへのダイレクト通知が使用されます。したがって、呼び出し元のタスクをブロックされた状態に配置するストリームバッファ API を呼び出すと、呼び出し元のタスクの通知状態と値が変更される可能性があります。

データの送信

xStreamBufferSend() は、タスク内のストリームバッファにデータを送信するために使用されます。xStreamBufferSendFromISR() は、割り込みサービスルーチン (ISR) 内のストリームバッファにデータを送信するために使用されます。

xStreamBufferSend() で、ブロック時間の指定が行えます。xStreamBufferSend() が呼び出され、ストリームバッファへの書き込みブロック時間が 0 以外になっている場合、バッファがいっぱいであると、メッセージバッファ領域が使用可能になるか、またはブロック時間が切れるまで、タスクはブロック状態になります。

sbSEND_COMPLETED() および sbSEND_COMPLETED_FROM_ISR() は、データがストリームバッファに書き込まれたときに (FreeRTOS API によって内部的に) 呼び出されるマクロです。更新されたストリームバッファのハンドルが必要です。これらのマクロはどちらも、データ待ちのストリームバッファにブロックされているタスクがあるかどうかを確認し、そのようなタスクが存在する場合はブロック状態からタスクを削除します。

このデフォルトの動作は、FreeRTOSConfig.h に sbSEND_COMPLETED() の独自の実装を提供することで変更できます。これは、ストリームバッファを使用してマルチコアプロセッサ上のコア間でデータを渡す場合に便利です。このシナリオでは、他の CPU コアに割り込みを生成するために sbSEND_COMPLETED() を実装することができ、割り込みのサービスルーチンは xStreamBufferSendCompletedFromISR() API を使用してデータを待機しているタスクをチェックし、必要に応じてブロックを解除します。

データの受信

xStreamBufferReceive() は、タスク内のストリームバッファからデータを読み込むために使用されます。xStreamBufferReceiveFromISR() は、割り込みサービスルーチン (ISR) 内のストリームバッファからデータを読み取るために使用されます。

xStreamBufferReceive() で、ブロック時間の指定が行えます。ストリームバッファから読み出すブロック時間が 0 以外で xStreamBufferReceive() が呼び出され、かつバッファが空の場合、指定された量のデータがストリームバッファで使用可能になるか、またはブロック時間が切れるまで、タスクはブロックされた状態になります。

タスクがブロック解除される前にストリームバッファ内に必要なデータの量は、ストリームバッファのトリガーレベルと呼ばれます。トリガーレベル 10 でブロックされたタスクは、少なくとも 10 バイトがバッファに書き込まれるか、タスクのブロック時間が切れるとブロック解除されます。トリガーレベルに達する前に読み出しタスクのブロック時間が終了すると、タスクはバッファに書き込まれたすべてのデータを受け取ります。タスクのトリガーレベルは、1 からストリームバッファサイズの間の値に設定する必要があります。xStreamBufferCreate() が呼び出されると、ストリームバッファのトリガーレベルが設定されます。また、xStreamBufferSetTriggerLevel() を呼び出すことで変更できます。

sbRECEIVE_COMPLETED() および sbRECEIVE_COMPLETED_FROM_ISR() は、ストリームバッファからデータを読み込むときに (FreeRTOS API によって内部的に) 呼び出されるマクロです。マクロは、ストリームバッファに領域が使用可能になるまで待機しているブロックされたタスクがあるかどうかを確認し、そのようなタスクがあれば、ブロック状態から削除します。sbRECEIVE_COMPLETED() のデフォルトの動作は、FreeRTOSConfig.h に代替の実装を提供することで変更できます。

メッセージバッファ

メッセージバッファでは、可変長の個別メッセージを割り込みサービスルーチンからタスクに、またはあるタスクから別のタスクに渡すことができます。たとえば、長さが 10、20 および 123 バイトのメッセージは、すべて同じメッセージバッファに書き込まれ、そこから読み取られます。10 バイトのメッセージは、個々のバイトではなく、10 バイトのメッセージとしてのみ読み取ることができます。メッセージバッファはストリームバッファの実装に基づいています。プロジェクトに stream_buffer.c ソースファイルを含めることで、メッセージバッファ機能を有効にすることができます。

メッセージバッファは、バッファ (ライター) に書き込むタスクまたは割り込みが 1 つだけであり、バッファ (リーダー) から読み取るタスクまたは割り込みが 1 つしかないことを前提としています。

ライターとリーダーが異なるタスクになることやサービスルーチンを中断することは安全と言えますが、複数のライターやリーダーがあるのは安全とは言えません。

メッセージバッファの実装では、タスクへのダイレクト通知が使用されます。したがって、呼び出し元のタスクをブロックされた状態に配置するストリームバッファ API を呼び出すと、呼び出し元のタスクの通知状態と値が変更される可能性があります。

メッセージバッファが可変サイズのメッセージを処理できるようにするため、メッセージバッファの前に各メッセージの長さがメッセージ自体に書き込まれます。長さは、size_t 型の変数に格納されます。これは、32 バイトのアーキテクチャでは通常 4 バイトです。したがって、メッセージバッファに 10 バイトのメッセージを書き込むと、実際には 14 バイトのバッファスペースが消費されます。同様に、メッセージバッファに 100 バイトのメッセージを書き込むと、実際には 104 バイトのバッファスペースが使用されます。

データの送信

xMessageBufferSend() は、タスクからメッセージバッファにデータを送信するために使用されます。xMessageBufferSendFromISR() は、割り込みサービスルーチン (ISR) からメッセージバッファにデータを送信するために使用されます。

xMessageBufferSend() で、ブロック時間の指定が行えます。メッセージバッファへ書き込むブロック時間が 0 以外で xMessageBufferSend() が呼び出され、かつバッファがいっぱいの場合、スペースがメッセージバッファ内で使用可能になるか、またはブロック時間が切れるまで、タスクはブロックされた状態になります。

sbSEND_COMPLETED() および sbSEND_COMPLETED_FROM_ISR() は、データがストリームバッファに書き込まれたときに (FreeRTOS API によって内部的に) 呼び出されるマクロです。これは更新されたストリームバッファのハンドルである単一のパラメータをとります。これらのマクロはどちらも、データ待ちのストリームバッファにブロックされているタスクがあるかどうかを確認し、存在する場合、マクロはブロック状態からタスクを削除します。

このデフォルトの動作は、FreeRTOSConfig.h に sbSEND_COMPLETED() の独自の実装を提供することで変更できます。これは、ストリームバッファを使用してマルチコアプロセッサ上のコア間でデータを渡す場合に便利です。このシナリオでは、他の CPU コアに割り込みを生成するために sbSEND_COMPLETED() を実装することができ、割り込みのサービスルーチンは xStreamBufferSendCompletedFromISR() API を使用してデータを待機していたタスクをチェックし、必要に応じてブロックを解除します。

データの受信

`xMessageBufferReceive()` は、タスク内のメッセージバッファからデータを読み込むために使用されます。`xMessageBufferReceiveFromISR()` は、割り込みサービスルーチン (ISR) 内のメッセージバッファからデータを読み取るために使用されます。`xMessageBufferReceive()` は、ブロック時間を指定できるようにします。メッセージバッファから読みだすブロック時間が 0 以外で `xMessageBufferReceive()` が呼び出され、かつバッファが空の場合、データが使用可能になるか、またはブロック時間が切れるまで、タスクはブロックされた状態になります。

`sbRECEIVE_COMPLETED()` および `sbRECEIVE_COMPLETED_FROM_ISR()` は、ストリームバッファからデータを読み込むときに (FreeRTOS API によって内部的に) 呼び出されるマクロです。マクロは、ストリームバッファに領域が使用可能になるまで待機しているブロックされたタスクがあるかどうかを確認し、そのようなタスクがあれば、ブロック状態から削除します。`sbRECEIVE_COMPLETED()` のデフォルトの動作は、`FreeRTOSConfig.h` に代替の実装を提供することで変更できます。

対称型マルチプロセッシング (SMP) のサポート

FreeRTOS カーネルでの SMP サポートにより、FreeRTOS カーネルの 1 つのインスタンスで複数の同じプロセッサコアにわたるタスクをスケジュールできます。コアアーキテクチャは同じで、同じメモリを共有する必要があります。

追加の API を除き、FreeRTOS API は、シングルコアバージョンと SMP バージョンで実質的に異なる点はありません。そのため、FreeRTOS シングルコアバージョン用に作成されたアプリケーションは、ほとんど変更することなく、SMP バージョンでコンパイルできます。ただし、シングルコアアプリケーションでは当てはまる前提がマルチコアアプリケーションでは当てはまらない可能性があるため、機能上の問題が発生する場合があります。

一般的な前提の 1 つは、優先度の高いタスクが実行されている間は、優先度の低いタスクを実行できないというものです。これはシングルコアシステムでは当てはまりましたが、複数のタスクを同時に実行できるため、マルチコアシステムには当てはまりません。アプリケーションが相互排他を実現するために相対的なタスクの優先順位に依存している場合、マルチコア環境で予期しない結果が発生する可能性があります。

もう 1 つの一般的な前提は、ISR を相互にまたは他のタスクと同時に実行できないということです。これは、マルチコア環境では当てはまりません。アプリケーション作成者は、タスクと ISR 間で共有されるデータにアクセスするときは、適切な相互排他を確保する必要があります。

ソフトウェアタイマー

ソフトウェアタイマーは、設定された時間に機能が実行されるようにします。タイマーによって実行される関数は、タイマーのコールバック関数と呼ばれます。タイマーが開始されてからコールバック関数が実行されるまでの時間をタイマーの周期と呼びます。FreeRTOS カーネルは以下のように、効率的なソフトウェアタイマー実装を提供します。

- 割り込みコンテキストからはタイマーコールバック関数を実行しません。
- タイマーが実際に切れない限り、処理時間は消費されません。
- ティック割り込みに処理オーバーヘッドを追加することはありません。
- 割り込みを無効にしている間は、リンクリストの構造を処理しません。

低電力サポート

ほとんどの組込みオペレーティングシステムと同様に、FreeRTOS カーネルは、時間を測定するために使用する、周期的なティック割り込みを生成するためにハードウェアタイマーを使用します。通常のハードウェアタイマー実装は、ティック割り込みを処理するために低電力状態を定期的に終了し、再び低電力状態にする必要があるため、省電力性に限りがあります。ティック割り込みの周期が早すぎる場合、ティックごとの低電力状態の開始/終了時に電力と時間が消費されるため、最低レベルの省電力モードを除くすべての省電力モードで、上限ゲインを上回ることなります。

この制限に対処するために、FreeRTOS には低電力アプリケーション用のティックレスタイマーモードがあります。FreeRTOS のティックレスアイドルモードは、アイドル期間 (実行可能なアプリケーションタスクがない期間) に周期ティック割り込みを停止し、ティック割り込みが再開されたときに RTOS ティックカウント値を修正します。ティック割り込みを停止すると、割り込みが発生するか、RTOS カーネルがタスクを準備完了状態に移行するまで、マイクロコントローラはディープレベルの省電力状態になります。

カーネルを設定する (FreeRTOSConfig.h)

`FreeRTOSConfig.h` ヘッダーファイルを使用して、特定のボードおよびアプリケーション用に FreeRTOS カーネルを設定できます。カーネル上にビルドされたすべてのアプリケーションでは、そのプリプロセッサインクルードパスに `FreeRTOSConfig.h` ヘッダーファイルが必要です。`FreeRTOSConfig.h` は、アプリケーションに固有であり、FreeRTOS カーネルのいずれかのソースコードディレクトリにではなく、アプリケーションディレクトリに配置する必要があります。

FreeRTOS デモおよびテストアプリケーション用の FreeRTOSConfig.h ファイルは *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/FreeRTOSConfig.h と *freertos*/vendors/*vendor*/boards/*board*/aws_tests/config_files/FreeRTOSConfig.h にあります。

FreeRTOSConfig.h に指定できる設定パラメータのリストについては、FreeRTOS.org を参照してください。

AWS IoT 埋め込み C 用の Device SDK

> **Note**
>
> この SDK は、経験豊富な組み込みソフトウェアデベロッパーによる使用を想定しています。

AWS IoT Device SDK for Embedded C (C-SDK) は、MIT オープンソースライセンスに基づく C ソースファイルのコレクションで、IoT デバイスを安全に接続するために組み込みアプリケーションで使用できます AWS IoT Core。これには、MQTT クライアント、HTTP クライアント、JSON パーサー、 AWS IoT Device Shadow、 AWS IoT Jobs、 AWS IoT フリートプロビジョニング、 AWS IoT Device Defender ライブラリが含まれます。この SDK はソース形式で配布され、アプリケーションコード、その他のライブラリ、および任意のオペレーティングシステム (OS) とともにお客様のファームウェアに組み込まれることが意図されています。

は通常 AWS IoT Device SDK for Embedded C 、最適化された C 言語ランタイムを必要とするリソース制約のあるデバイスを対象としています。この SDK は、任意のオペレーティングシステムで使用でき、任意のプロセッサタイプ (MCU や MPU など) でホストできます。ただし、デバイスに十分なメモリと処理リソースがある場合は、上位のいずれかの AWS IoT Device SDK を使用することをお勧めします。

詳細については次を参照してください:

- AWS IoT 埋め込み C 用の Device SDK
- AWS IoT GitHub の埋め込み C 用 Device SDK
- AWS IoT 埋め込み C Readme 用の Device SDK
- AWS IoT Device SDK for Embedded C サンプル

FreeRTOS 共通 IO APIs

共通 IO API は、ドライバーと上位レベルのアプリケーションコード間の共通のインターフェイスを提供するハードウェア抽象化レイヤー (HAL) として機能します。FreeRTOS Common IO は、サポートされているリファレンスボード上の共通シリアルデバイスにアクセスするための標準 API セットを提供します。これらの API の実装は含まれていません。これらの共通 API は、これらの周辺機器と通信、相互作用し、プラットフォーム間でコードを機能させることができます。共通 IO がない場合、低レベルデバイスで動作するコードの記述は、シリコンベンダー固有です。

> **Note**
>
> FreeRTOS は、機能するために Common IO API の実装を必要としませんが、ベンダー固有の API の代わりに、マイクロコントローラーベースのボード上の特定の周辺機器とインターフェイスする方法として Common IO API を使用しようとします。

一般に、デバイスドライバーは基盤となるオペレーティングシステムに依存せず、特定のハードウェア構成に固有です。HAL は、特定のドライバーの動作の詳細を抽象化し、そのようなデバイスを制御するための統一された API を提供します。同じ API を使用して、複数のマイクロコントローラー (MCU) ベースのリファレンスボード間でさまざまなデバイスドライバーにアクセスできます。

[Libraries] (ライブラリ)

現在、FreeRTOS は、共通 IO - ベーシックと共通 IO - BLE の 2 つの共通 IO ライブラリを提供しています。

共通 IO - ベーシック

概要

共通 IO - ベーシックは、MCU ベースのボードに搭載されている基本的な I/O 周辺機器や機能を処理する API を提供します。共通 IO - ベーシックのリポジトリは GitHub で公開されています。

サポートされている周辺機器

- ADC
- GPIO

- I2C
- PWM
- SPI
- UART
- Watchdog
- Flash
- RTC
- EFUSE
- Resets
- I2S
- パフォーマンスカウンター
- ハードウェアプラットフォーム情報

サポートされている機能

- 同期読み取り/書き込み

  要求された量のデータが転送されるまで、関数は戻りません。

- 非同期読み取り/書き込み

  関数はすぐに戻り、データ転送は非同期的に行われます。アクションが完了すると、登録されたユーザーコールバックが呼び出されます。

周辺機器固有

- I2C

  複数のオペレーションを 1 つのトランザクションに結合します。1 つのトランザクションで書き込みおよび読み取りアクションを実行するために使用されます。

- SPI

  プライマリとセカンダリの間でデータを転送します。つまり、書き込みと読み取りが同時に行われます。

API リファレンス

完全な API リファレンスについては、「Common IO - basic API reference」をご覧ください。

共通 IO - BLE

概要

共通 IO - BLE は、製造元の Bluetooth Low Energy スタックからの抽象化を提供します。デバイスの制御、GAP および GATT 操作の実行に使用できる以下のインターフェイスを提供します。共通 IO - BLE のリポジトリは GitHub で公開されています。

Bluetooth デバイスマネージャー:

Bluetooth デバイスの制御、デバイス検出操作、その他の接続関連タスクを実行するためのインターフェイスを提供します。

BLE アダプターマネージャー:

BLE 固有の GAP API 関数のインターフェイスを提供します。

Bluetooth クラシックアダプターマネージャー:

デバイスの BT クラシック機能を制御するためのインターフェイスを提供します。

GATT サーバー:

Bluetooth GATT サーバー機能を使用するためのインターフェイスを提供します。

GATT クライアント:

Bluetooth GATT クライアント機能を使用するためのインターフェイスを提供します。

A2DP 接続インターフェイス:

ローカルデバイスの A2DP ソースプロファイル用のインターフェイスを提供します。

API リファレンス

完全な API リファレンスについては、「Common IO - BLE API reference」をご覧ください。

Amazon 共通ソフトウェア用の共通 IO

共通 IO API は、デバイス用 Amazon 共通ソフトウェアに必要な実装の一部であり、特にベンダーのデバイス移植キット (DPK) に実装する必要があります。

ACS とは

デバイス用 Amazon 共通ソフトウェア (ACS) は、Amazon デバイス SDK をデバイスにすばやく統合できるようにするソフトウェアです。ACS は、接続、デバイス移植キット (DPK)、多層テストス

イートなどの一般的な機能のための、統一された API 統合レイヤー、事前検証済みでメモリ効率の高いコンポーネントを提供します。

認定プログラム

「Amazon Common Software for Devices」の認定プログラムでは、特定のマイクロコントローラーベースの開発ボード上で動作する ACS DPK (Device Porting Kit) のビルドが、プログラムが公開しているベストプラクティスと互換性があり、認定プログラムで指定された ACS が義務付けているテストに合格できるほど堅牢であることを検証します。

このプログラムの対象となるベンダーは、「ACS Chipset Vendors」のページに記載されています。

認定の詳細については、「ACS for Devices」を参照してください。

FreeRTOS の使用を開始する

トピック

- Quick Connect の使用を開始する
- FreeRTOS ライブラリの詳細を理解する
- 安全で堅牢な AWS IoT 製品を構築する
- AWS IoT アプリケーションを開発する

Quick Connect の使用を開始する

すばやく調べるには AWS IoT、AWS クイック接続デモから始めます。Quick Connect デモはセットアップが簡単で、パートナーが提供する FreeRTOS 認定ボードを AWS IoT に接続します。

AWS IoT と AWS IoT コンソールの詳細については、AWS IoT 「開始方法」チュートリアルを参照してください。選択したボードのビルドシステムとツールを使用して、Quick Connect デモで提供されるデモソースコードを変更して、 AWS アカウントに接続できます。 AWS IoT アカウントのコンソールからデータフローが表示されるようになりました。

FreeRTOS ライブラリの詳細を理解する

IoT デバイスと がどのように AWS IoT 連携するかを理解したら、FreeRTOS ライブラリと長期サポート (LTS) ライブラリの探索を開始できます。

FreeRTOS ベースの AWS IoT デバイス用に一般的に使用されるライブラリは次のとおりです。

- FreeRTOS カーネル
- coreMQTT
- AWS IoT Over-the-Air通信 (OTA)

ライブラリ固有の技術文書やデモについては、freertos.org にアクセスしてください。

安全で堅牢な AWS IoT 製品を構築する

IoT デバイスソフトウェアをより安全かつ堅牢にするためのベストプラクティスについては、「注目の FreeRTOS AWS IoT 統合」を参照してください。これらの FreeRTOS IoT 統合は、FreeRTOS ソ

フトウェアと、ハードウェアセキュリティ機能を備えたパートナー提供のボードを組み合わせてセキュリティを強化するように設計されています。本番環境では、そのまま使用することも、独自の設計のモデルとして使用することもできます。

AWS IoT アプリケーションを開発する

AWS IoT 製品のアプリケーションプロジェクトを作成するには、次の手順に従います。

1. freertos.org から最新の FreeRTOS または長期サポート (LTS) バージョンをダウンロードするか、FreeRTOS-LTS GitHub リポジトリからクローンを作成します。可能であれば、MCU ベンダーのツールチェーンから必要な FreeRTOS ライブラリをプロジェクトに統合することもできます。
2. 「FreeRTOS Porting guide」に従って、プロジェクトを作成し、開発環境をセットアップし、FreeRTOS ライブラリをプロジェクトに統合します。FreeRTOS-Libraries-Integration-Tests GitHub リポジトリを使用して移植を検証します。

AWS IoT Device Tester FreeRTOS 用

IDT for FreeRTOS は、FreeRTOS オペレーティングシステムでのデータスループットレートを評価するためのツールです。デバイステスター (IDT) は、最初にデバイスへの USB 接続または UART 接続を開きます。次に、さまざまな条件下でデバイスの機能をテストするように設定された FreeRTOS のイメージをフラッシュします。 AWS IoT Device Tester スイートは拡張可能で、IDT はお客様の AWS IoT テストオーケストレーションに使用されます。

IDT for FreeRTOS は、テスト対象のデバイスに接続されているホストコンピュータ (Windows、macOS、または Linux) 上で動作します。IDT はテストケースの設定とオーケストレーションを行い、結果を集計します。また、テストの実行を管理するためのコマンドラインインターフェイスも用意されています。

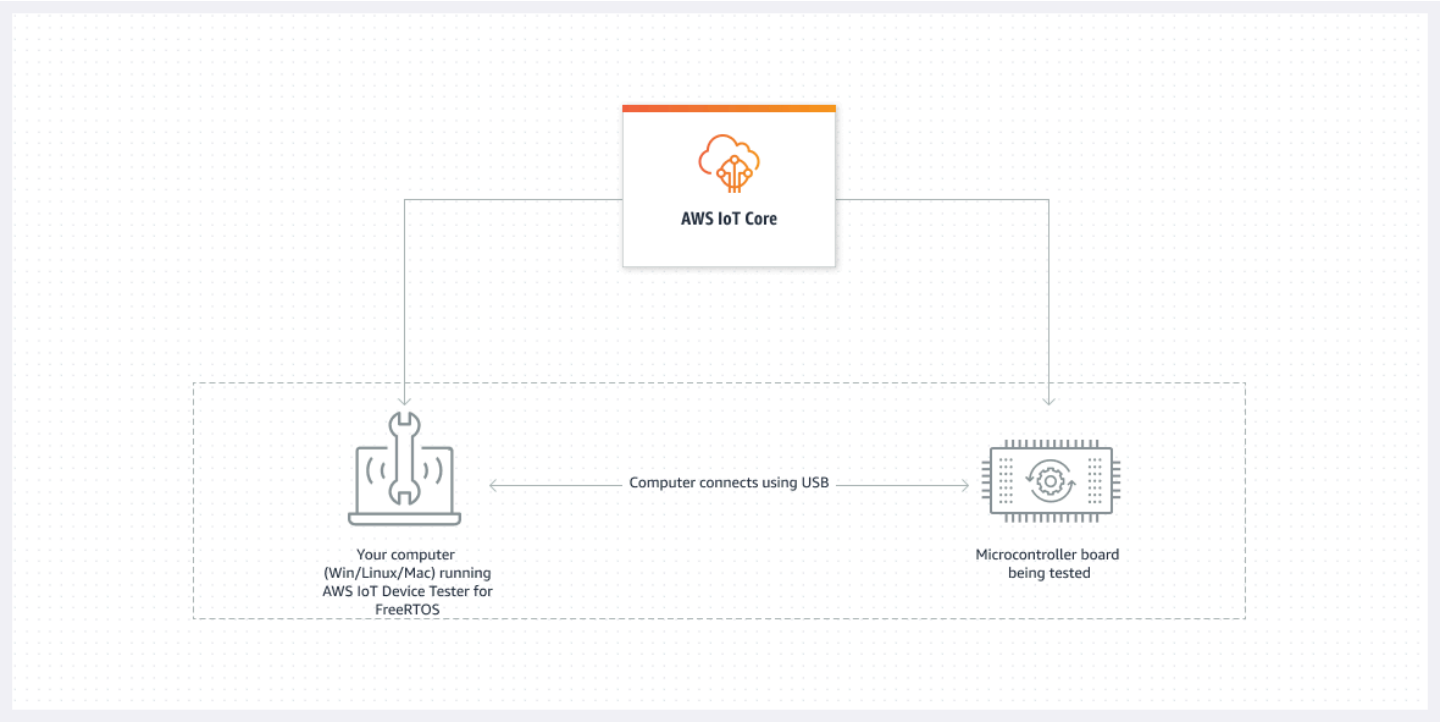
FreeRTOS 認定スイート

IDT for FreeRTOS は、マイクロコントローラー上の FreeRTOS のポートを検証し、信頼性が高く安全な方法で AWS IoT と効果的に通信できるかどうかを検証します。具体的には、FreeRTOS ライブラリの移植レイヤーインターフェイスが正しく実装されているかを検証します。また、 を使用して end-to-endのテストも実行します AWS IoT Core。例えば、ボードで MQTT メッセージを送受信して正しく処理できるかを検証します。

FreeRTOS 認定 (FRQ) 2.0 スイートでは、「FreeRTOS Qualification Guide」で定義されている FreeRTOS-Libraries-Integration-Tests と Device Advisor のテストケースを使用します。

IDT for FreeRTOS はテストレポートを生成します。テストレポートは AWS 、 Partner Network (APN) に送信して、FreeRTOS デバイスを AWS Partner Device Catalog に含めることができます。詳細については、AWS デバイス認定プログラムを参照してください。

次の図に、FreeRTOS 認定のテストインフラストラクチャのセットアップを示します。



IDT for FreeRTOS は、テストリソースをテストスイートとテストグループに整理します。

- テストスイートは、デバイスが FreeRTOS の特定のバージョンで動作することを確認するために使用されるテストグループのセットです。
- テストグループは、BLE や MQTT メッセージングなど、特定の機能に関連する個々のテストケースのセットです。

詳細については、「テストスイートのバージョン」を参照してください

カスタムテストスイートを理解する

IDT for FreeRTOS では、標準化された構成設定および結果形式とテストスイート環境が統合されています。この環境では、ご使用のデバイスやデバイスソフトウェア用にカスタムテストスイートを開発できます。独自の内部検証用のカスタムテストを追加したり、デバイス検証のためにこれらのテストを顧客に提供したりできます。

カスタムテストスイートの設定方法によって、カスタムテストスイートを実行するためにユーザーに提供する必要がある設定構成が決まります。詳細については、「独自の IDT テストスイートを開発して実行する」を参照してください。

のサポートされているバージョン AWS IoT Device Tester

このトピックでは、 AWS IoT Device Tester for FreeRTOS のサポートされているバージョンを示します。ベストプラクティスとして、FreeRTOS のターゲットバージョンをサポートする IDT for FreeRTOS の最新バージョンを使用することをお勧めします。IDT for FreeRTOS の各バージョンには、サポートする FreeRTOS の対応するバージョンが 1 つ以上あります。IDT for FreeRTOS の新しいバージョンのリリース時には、IDT for FreeRTOS の新しいバージョンのダウンロードをお勧めします。

ソフトウェアをダウンロードすることで、ダウンロードアーカイブに含まれる AWS IoT Device Tester ライセンス契約に同意したものとみなされます。

> **Note**
>
> AWS IoT Device Tester for FreeRTOS を使用する場合は、最新の FreeRTOS-LTS バージョンの最新のパッチリリースに更新することをお勧めします。

> **Important**
>
> 2022 年 10 月現在、 AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 は署名付き認定レポートを生成しません。IDT FRQ 1.0 バージョンを使用するAWS デバイス認定プログラムを通じて、新しい AWS IoT FreeRTOS デバイスを AWS Partner Device Catalog に一覧表示する資格を得ることはできません。IDT FRQ 1.0 を使用して FreeRTOS デバイスを認定することはできませんが、引き続き FRQ 1.0 を使用して FreeRTOS デバイスをテストすることはできます。FreeRTOS デバイスを認定し、AWS Partner Device Catalog にリストする際には、IDT FRQ 2.0 を使用することが推奨されます。

AWS IoT Device Tester for FreeRTOS の最新バージョン

最新バージョンの IDT for FreeRTOS をダウンロードするには、以下のリンクを使用します。

AWS IoT Device Tester for FreeRTOS の最新バージョン

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
IDT v4.9.0	FRQ_2.5.0	• 202112.00 • 202212.00 • 202212.01 • FreeRTOS LTS ライブラリを使用する FreeRTOS 202210-LTS のすべてのパッチ。	• Linux • macOS • Windows	2023.04.04	• FreeRTOS 202112、202212、、および FreeRTOS ライブラリを使用する FreeRTOS 202210-LTS のすべてのパッチに対するテストをサポートします。詳細については、「README.md」を参照してください。FreeRTOS-LTS のパッチバージョンを `manifest.yml` に含める必要があります。 • OTA E2E テストの実行時間が改

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
					善されました。 • device.json に記載されているデバイスの数を 1 つに制限します。 • 小さなバグ修正と機能向上。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。このように実行すると、クラッシュまたはデータの破損が発生する可能性があります。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。

IDT for FreeRTOS の以前のバージョン

以下の IDT for FreeRTOS の以前のバージョンもサポートされます。

AWS IoT Device Tester for FreeRTOS の以前のバージョン

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
IDT v4.8.1	FRQ_2.4.0	• 202112.00 • 202212.00 • 202212.01 • FreeRTOS LTS ライブラリを使用する FreeRTOS 202210-LTS のすべてのパッチ。	• Linux • macOS • Windows	2023.01.23	• 詳細については、「README.md」を参照してください。FreeRTOS-LTS のパッチバージョンを `manifest.yml` に含める必要があります。 • 小さなバグ修正と機能向上。
IDT v4.6.0	FRQ_2.3.0	• 202112.00 • 202212.00 • 202212.01 • FreeRTOS LTS ライブラリを使用する 202210-LTS。	• Linux • macOS • Windows	2022.11.16	• 詳細については、「README.md」を参照してください。FreeRTOS-LTS のパッチバージョンを `manifest.yml` に含める必要があります。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
					• FreeRTOS 202210-LTS リリースに含まれる内容の詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • ウェブベースのユーザーインターフェイスを介して FreeRTOS AWS IoT Device Tester の設定と実行を行う機能を追加しました。開始するには、IDT for FreeRTOS 認定スイート 2.0 (FRQ 2.0)

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
					の UIを参照してください。 • 実行時に作成され使用されたソースコードの変更されたコピーを、テスト後のデバッグ用に保持するオプションを追加します。詳細については「ビルド、フラッシュ、テスト設定を設定する」を参照してください。 • IDT クライアント SDK の Java サポートを追加します。IDT クライアント

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
					SDK の詳細については、「独自の IDT テストスイートを開発して実行する」を参照してください。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	ダウンロードリンク	リリース日	リリースノート
IDT v4.5.11	FRQ_2.2.0	• 202112.00 • 202212.00 • 202212.01 • FreeRTOS LTS ライブラリを使用する 202210-LTS。	• Linux • macOS • Windows	2022.10.14	• 詳細については、「README.md」を参照してください。FreeRTOS-LTS のパッチバージョンを `manifest.yml` に含める必要があります。 • FreeRTOS 202210-LTS リリースに含まれる内容の詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • 小さなバグ修正と機能向上。

詳細については、「のサポートポリシーを理解する AWS IoT Device Tester」を参照してください。

IDT for FreeRTOS のサポートされていないバージョン

このセクションでは、IDT for FreeRTOS のサポートされていないバージョンを示します。サポートされていないバージョンのバグ修正や更新プログラムは受けられません。詳細については、のサポートポリシーを理解する AWS IoT Device Tester を参照してください。

IDT-FreeRTOS の以下のバージョンはサポートされなくなりました。

AWS IoT Device Tester for FreeRTOS のサポートされていないバージョン

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.5.10	FRQ_2.1.4	• 202112.00 • FreeRTOS LTS ライブラリを使用する 202012-LTS。	2022.09.02	• FreeRTOS 202012-LTS リリースに含まれる内容の詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • `OTA End to End` テストグループに影響する問題が解決されました。 • 認定ランの実行から `FullTransportInterfacePlainText` が除去

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				されました。プレーンテキストは、`-\-group-id` フラグを使用することで、引き続き開発テストグループとして実行できます。 • コンソールとファイルの出力のロギングと可読性が向上しました。 • 小さなバグ修正と機能向上。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.5.9	FRQ_2.1.3	• 202112.00 • FreeRTOS LTS ライブラリを使用する 202012.04-LTS。	2022.08.17	• FreeRTOS 202012.04-LTS リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • `FreeRTOSIntegrity` テストグループに影響する問題が解決されました。 • `FullCloudIoT` テストグループが更新され、「MQTT Connect Exponential Backoff Retries」テストケースが削除されました。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• 小さなバグ修正と機能向上。
IDT v4.5.6	FRQ_2.1.2	• 202112.00 • FreeRTOS LTS ライブラリを使用する 202012.04-LTS。	2022.06.29	• FreeRTOS 202012.04-LTS リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • AWS IoT Core Device Advisor に対してボードをテストする、新しいテストグループ `FullCloudIoT` が追加されました。 • OTA E2E テストケースに影響する問題が解決されました。 • 小さなバグ修正と機能向上。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.5.5	FRQ_2.1.1	• 202112.00 • FreeRTOS LTS ライブラリを使用する 202012.04-LTS。	2022.06.06	• FreeRTOS 202012.04-LTS リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • AWS IoT Core Device Advisor に対してボードをテストする、新しいテストグループ `FullCloudIoT` が追加されました。 • FreeRTOSVersion テストケースと FreeRTOSIntegrity テストケースに影響する問題が解決されました。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• 小さなバグ修正と機能向上。
IDT v4.5.5	FRQ_2.1.0	• 202107.00 • 202112.00 • FreeRTOS LTS ライブラリを使用する 202012.04-LTS。	2022.05.31	• FreeRTOS 202012.04-LTS リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • AWS IoT Core Device Advisor に対してボードをテストする、新しいテストグループ `FullCloudIoT` が追加されました。 • 小さなバグ修正と機能向上。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.5.4	FRQ_2.0.0	• 202112.00 • FreeRTOS LTS ライブラリを使用する 202012.04-LTS。	2022.05.09	• FreeRTOS 202012.04-LTS リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • `aws/amazon-freertos` GitHub リポジトリで提供されている Amazon FreeRTOS のバージョンのみを使用してボードを認定する要件が削除されました。 • 小さなバグ修正と機能向上。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.5.2	FRQ_1.6.2	202107.00	2022.01.25	• FreeRTOS 202107.00 リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • カスタムテストスイートを設定するための新しい IDT テストオーケストレーターを実装しました。詳しくは、「IDT テストオーケストレーターを設定する」を参照してください。 • 小さなバグ修正と機能向上。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.0.3	FRQ_1.5.1	202012.00	2021.07.30	• ハードウェアセキュリティモジュールでロックダウンされた認証情報を使用するデバイスの認定がサポートされます。 • 小さなバグ修正と機能向上。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.3.0	FRQ_1.6.1	202107.00	2021.07.26	• FreeRTOS 202107.00 リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • ウェブベースのユーザーインターフェイスを介して FreeRTOS AWS IoT Device Tester の設定と実行を行う機能を追加しました。開始するには、IDT ユーザーインターフェイスを使用して FreeRTOS 認定スイートを実行するを参照してください。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート

IDT v4.1.0	FRQ_1.6.0	202107.00	2021.07.21	• FreeRTOS 202107.00 リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。
				• OTA 認定から次のテストケースが削除されました。
				  • OTA エージェント
				  • OTA Missing Filename (OTA ファイル名の消失)
				  • OTA Max Configured Number of Blocks (OTA ブロックの最大設定数)
				• OTA 認定から OTA データプレーンの Both テス

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				トグループが削除されました。device.json ファイルの OTADataPlaneProtocol 設定の値は、HTTP と MQTT のみが使用できます。 • FreeRTOS ソースコードの変更に対応して、userdata.json ファイルの freertosFileConfiguration 設定に以下の変更が実装されました。 • otaAgentTestsConfig と otaAgentDemosConfig に指定するファイル名が

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				aws_ota_agent_config.h から ota_config.h に変更されました。 • ファイルパスを新しい ota_demo_config.h ファイルに指定するための新しい otaDemosConfig オプション設定が追加されました。 • デバイスをフラッシュして FreeRTOS テストグループを実行する時間とテストの実行を開始する時間との間に遅延を指定するための新しいフィールド testStartDelayms を

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				`userdata.json` に追加しました。値はミリ秒単位で指定します。この遅延により、IDT が接続する時間ができ、テスト出力を失うことはありません。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v4.0.1	FRQ_1.4.1	202012.00	2021.01.19	• FreeRTOS 202012.00 リリースに含まれる内容の詳細については、GitHub の CHANGELOG.md ファイルを参照してください。 • OTA (無線通信経由) E2E (エンドツーエンド) テストケースが追加されました。 • FreeRTOS LTS ライブラリを使用する FreeRTOS 202012.00 を実行する開発ボードの認定がサポートされます。 • セルラー接続を使用する FreeRTOS 開発ボードの認定サポートが追加されました。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• エコーサーバー構成のバグが修正されました。 • AWS IoT Device Tester for FreeRTOS を使用して、独自のカスタムテストスイートを開発および実行できるようになりました。詳細については、独自の IDT テストスイートを開発して実行するを参照してください。 • コード署名された IDT アプリケーションを提供するため、Windows または macOS で実行するときにアクセス許可を付与する必要はありません。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• BLE テスト結果の解析ロジックを改善しました。
IDT v3.4.0	FRQ_1.3.0	202011.01	2020.11.05	• 詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • 「RSA」が有効な PKCS11 構成オプションではないバグが修正されました。 • OTA テスト後に Amazon S3 バケットが正しくクリーンアップされないバグが修正されました。 • FullMQTT テストグループ内の新しいテストケースをサポートするための更新。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v3.3.0	FRQ_1.2.0	202007.00	2020.09.17	• 詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • 無線通信 (OTA) 更新の一時停止および再開機能を検証するための新しいエンドツーエンドテスト。 • eu-central-1 リージョンのユーザーが OTA テストの設定検証に合格できないバグが修正されました。 • `run-suite` コマンドに `--update-idt` パラメータが追加されました。このオプションを使用すると、IDT の更新プロンプトへの応答

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				を設定できます。 • `run-suite` コマンドに `--update-managed-policy` パラメータが追加されました。このオプションを使用すると、マネージドポリシーの更新プロンプトへの応答を設定できます。 • 以下を含む内部の機能向上とバグの修正: • テストスイートの自動更新で、設定ファイルのアップグレードが改善されました。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v3.0.2	FRQ_1.0.1	202002.00		• 詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • IDT 内のテストスイートの自動更新を追加します。これで、IDT は、お使いの FreeRTOS バージョンで利用できる最新のテストスイートをダウンロードできるようになります。この機能を使用すると、次の操作を実行できます。 • `upgrade-test-suite` コマンドを使用して、最新のテストスイートを

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				ダウンロードします。 • IDT の起動時にフラグを設定して、最新のテストスイートをダウンロードします。 -u *flag* オプションを使用します。常にダウンロードするには *flag* を「y」にし、既存のバージョンを使用するには「n」にします。 利用可能なテストスイートのバージョンが複数ある場合は、IDT の起動時にテストス

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				イート ID を指定しない限り、最新バージョンが使用されます。 • 新しい `list-supported-versions` オプションを使用して、インストールされているバージョンの IDT でサポートされている FreeRTOS およびテストスイートのバージョンを一覧表示します。 • グループにテストケースを一覧表示し、個々のテストを実行します。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				テストスイートは、1.0.0 から始まる major.minor.patch 形式でバージョン管理されます。 • list-supported-products コマンドが追加されます。インストールされている IDT のバージョンでサポートされている FreeRTOS およびテストスイートのバージョンを一覧表示できます。 • list-test-cases コマンドが追加されます。テストグループで使用できるテストケースを一覧表示できます。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• `test-id` コマンドに `run-suite` オプションが追加されます。このオプションを使用して、テストグループ内の個々のテストケースを実行できます。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v1.7.1	FRQ_1.0.0	202002.00		• 詳細については、GitHub の「CHANGELOG.md」ファイルを参照してください。 • 無線通信経由 (OTA) のエンドツーエンドテストケース用のカスタムコード署名方法をサポートするため、独自のコード署名コマンドとスクリプトを使用して OTA ペイロードに署名できます。 • テストの開始前にシリアルポートの事前チェックを追加します。`device.json` ファイルでシリアルポートの設定が誤っている場合に、テス

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				トの失敗が迅速化され、エラーメッセージも改善されました。 • AWS IoT Device Tester を実行するために必要なアクセス許可が含まれている AWS 管理ポリシー `AWSIoTDeviceTesterForFreeRTOSFullAccess` が追加されました。新しいリリースで追加のアクセス許可が必要になった場合は、この管理ポリシーにアクセス許可が自動的に追加されるため、IAM のアクセス許可を手動で更新する必要はありません。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• 結果ディレクトリ内の `AFQ_Report.xml` というファイルの名前が `FRQ_Report.xml` になりました。
IDT v1.6.2	FRQ_1.0.0	202002.00		• FreeRTOS 開発ボードを認定するために、HTTPS を介した OTA のオプションのテストがサポートされます。 • テストで AWS IoT ATS エンドポイントをサポートします。 • テストスイートの開始前にユーザーに最新の IDT バージョンを通知する機能をサポートします。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v1.5.2	FRQ_1.0.0	201910.00		• セキュアエレメント (オンボードキー) を使用した FreeRTOS デバイスの認定がサポートされます。 • Secure Sockets および Wi-Fi テストグループの設定可能なエコーサーバーポートをサポートします。 • タイムアウトを増やすためのタイムアウト乗数フラグをサポートします。これは、タイムアウトに関連するエラーのトラブルシューティングを行うときに役立ちます。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
				• ログ解析のバグ修正を追加しました。 • テストで iot ats エンドポイントをサポートします。
IDT v1.4.1	FRQ_1.0.0	201908.00		• 新しい PKCS11 ライブラリとテストケースの更新のサポートを追加しました。 • 実行可能なエラーコードを導入しました。詳細については、「IDT エラーコード」を参照してください。 • IDT の実行に使用される IAM ポリシーを更新しました。

AWS IoT Device Tester version	テストスイートのバージョン	サポートされている FreeRTOS バージョン	リリース日	リリースノート
IDT v1.3.2	FRQ_1.0.0	201906.00		• Bluetooth Low Energy (BLE) のテストへのサポートが追加されました。 • IDT コマンドラインインターフェイス (CLI) コマンドのユーザーエクスペリエンスが向上しました。 • IDT の実行に使用される IAM ポリシーを更新しました。
IDT-FreeRTOS v1.2	FRQ_1.0.0	• FreeRTOS v1.4.8 • FreeRTOS v1.4.9		CMAKE ビルドシステムで FreeRTOS デバイスをテストするためのサポートが追加されました。
IDT-FreeRTOS v1.1	FRQ_1.0.0			
IDT-FreeRTOS v1.0	FRQ_1.0.0			

IDT for FreeRTOS のダウンロード

このトピックでは、IDT for FreeRTOS をダウンロードする際のオプションについて説明します。次のいずれかのソフトウェアダウンロードリンクを使用するか、指示に従ってプログラムで IDT をダウンロードできます。

> **Important**
>
> 2022 年 10 月現在、 AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 は署名付き認定レポートを生成しません。IDT FRQ 1.0 バージョンを使用するAWS デバイス認定プログラムを通じて、新しい AWS IoT FreeRTOS デバイスを AWS Partner Device Catalog に一覧表示する資格を得ることはできません。IDT FRQ 1.0 を使用して FreeRTOS デバイスを認定することはできませんが、引き続き FRQ 1.0 を使用して FreeRTOS デバイスをテストすることはできます。FreeRTOS デバイスを認定し、AWS Partner Device Catalog にリストする際には、IDT FRQ 2.0 を使用することが推奨されます。

トピック

- IDT を手動でダウンロードする
- IDT をプログラムでダウンロード

ソフトウェアをダウンロードすることで、ダウンロードアーカイブに含まれる AWS IoT Device Tester ライセンス契約に同意したものとみなされます。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。

IDT を手動でダウンロードする

このトピックでは、IDT for FreeRTOS のサポートされているバージョンを示します。ベストプラクティスとして、FreeRTOS のターゲットバージョン AWS IoT Device Tester をサポートする の最新バージョンを使用することをお勧めします。FreeRTOS の新しいリリースでは、 AWS IoT Device Testerの新しいバージョンのダウンロードが必要になる場合があります。を使用している FreeRTOS

のバージョンと互換性 AWS IoT Device Tester がない場合、テストランを開始すると通知を受け取ります。

「のサポートされているバージョン AWS IoT Device Tester」を参照してください。

IDT をプログラムでダウンロード

IDT には、プログラムで IDT をダウンロードできる URL の取得に使用できる API オペレーションが用意されています。この API オペレーションを使用して、IDT が最新バージョンであることを確認することもできます。この API オペレーションには、以下のエンドポイントがあります。

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

この API オペレーションを呼び出すには、**iot-device-tester:LatestIdt** アクションを実行するための権限が必要です。 AWS 署名をサービス名`iot-device-tester`として含める

API リクエスト

HostOS - ホストマシンのオペレーティングシステム。次のオプションから選択します。

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

testSuiteType - テストスイートのタイプ。次のオプションを選択します。

FR – IDT for FreeRTOS

ProductVersion

(オプション) FreeRTOS のバージョン。サービスは、そのバージョンの FreeRTOS と互換性のある IDT のバージョンのうち、最新のバージョンを返します。このオプションを指定しない場合、サービスは最新バージョンの IDT を返します。

API レスポンス

API レスポンスの形式は次のとおりです。`DownloadURL` には zip ファイルが付属しています。

```
{
    "Success": True or False,
    "Message": Message,
    "LatestBk": {
```

```
        "Version": The version of the IDT binary,
        "TestSuiteVersion": The version of the test suite,
        "DownloadURL": The URL to download the IDT Bundle, valid for one hour
    }
}
```

例

プログラムで IDT をダウンロードするには、以下の例を参照してください。これらの例では、AWS_ACCESS_KEY_ID に保存した認証情報および AWS_SECRET_ACCESS_KEY 環境変数が使用されます。セキュリティのベストプラクティスに従い、認証情報はコードに保存しないでください。

Example

例: cURL バージョン 7.75.0 以降を使用したダウンロード (Mac および Linux)

cURL バージョン 7.75.0 以降の場合、aws-sigv4 フラグを使用して API リクエストに署名できます。この例では、レスポンスからのダウンロード URL の解析に jq を使用します。

> **Warning**
>
> aws-sigv4 フラグでは、curl GET リクエストのクエリパラメータが HostOs/ProductVersion/TestSuiteType または HostOs/TestSuiteType の順序である必要があります。注文が一致しない場合、API ゲートウェイから正規文字列の署名が一致しないというエラーが発生します。
> オプションのパラメータ ProductVersion が含まれている場合は、「サポートされているAWS IoT Device Tester バージョンの for FreeRTOS」に記載されているように、サポートされている製品バージョンを使用する必要があります。

- *us-west-2* を に置き換えます AWS リージョン。リージョンコードの一覧については、「リージョンエンドポイント」を参照してください。
- *linux* をホストマシンのオペレーティングシステムに置き換えます。
- *202107.00* は、ご使用の FreeRTOS のバージョンに置き換えてください。

```
url=$(curl --request GET "https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&ProductVersion=202107.00&TestSuiteType=FR" \
```

```
--user $AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY \
--aws-sigv4 "aws:amz:us-west-2:iot-device-tester" \
| jq -r '.LatestBk["DownloadURL"]')

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

Example

例: 以前のバージョンの cURL を使用したダウンロード (Mac および Linux)

次の cURL コマンドは、署名して計算する AWS 署名とともに使用できます。署名に署名して計算する方法の詳細については AWS 、AWS 「 API リクエストの署名」を参照してください。

- *linux* をホストマシンのオペレーティングシステムに置き換えます。
- *#######*を **20220210T004606Z** などの日付と時刻に置き換えます。
- *##*を **20220210** などの日付に置き換えます。
- *AWSRegion* を に置き換えます AWS リージョン。リージョンコードの一覧については、「リージョンエンドポイント」を参照してください。
- *AWSSignature* を、ユーザーが生成する AWS 署名に置き換えます。

```
curl --location --request GET 'https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&TestSuiteType=FR' \
--header 'X-Amz-Date: Timestamp \
--header 'Authorization: AWS4-HMAC-SHA256 Credential=$AWS_ACCESS_KEY_ID/Date/AWSRegion/
iot-device-tester/aws4_request, SignedHeaders=host;x-amz-date, Signature=AWSSignature'
```

Example

例: Python スクリプトを使用したダウンロード

この例では Python リクエストライブラリを使用しています。この例は、 AWS 全般のリファレンスの AWS API リクエストに署名する Python の例から適用されています。

- *us-west-2* を、ご利用のリージョンに置き換えます。リージョンコードの一覧については、「リージョンエンドポイント」を参照してください。
- *linux* をホストマシンのオペレーティングシステムに置き換えます。

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

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

# Key derivation functions. See:
# http://docs.aws.amazon.com/general/latest/gr/signature-v4-examples.html#signature-v4-
examples-python
def sign(key, msg):
    return hmac.new(key, msg.encode('utf-8'), hashlib.sha256).digest()

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

# Read AWS access key from env. variables or configuration file. Best practice is NOT
# to embed credentials in code.
access_key = os.environ.get('AWS_ACCESS_KEY_ID')
secret_key = os.environ.get('AWS_SECRET_ACCESS_KEY')
if access_key is None or secret_key is None:
```

```
    print('No access key is available.')
    sys.exit()

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

# ************* TASK 1: CREATE A CANONICAL REQUEST *************
# http://docs.aws.amazon.com/general/latest/gr/sigv4-create-canonical-request.html
# Step 1 is to define the verb (GET, POST, etc.)--already done.
# Step 2: Create canonical URI--the part of the URI from domain to query
# string (use '/' if no path)
canonical_uri = '/latestidt'
# Step 3: Create the canonical query string. In this example (a GET request),
# request parameters are in the query string. Query string values must
# be URL-encoded (space=%20). The parameters must be sorted by name.
# For this example, the query string is pre-formatted in the request_parameters
 variable.
canonical_querystring = request_parameters
# Step 4: Create the canonical headers and signed headers. Header names
# must be trimmed and lowercase, and sorted in code point order from
# low to high. Note that there is a trailing \n.
canonical_headers = 'host:' + host + '\n' + 'x-amz-date:' + amzdate + '\n'
# Step 5: Create the list of signed headers. This lists the headers
# in the canonical_headers list, delimited with ";" and in alpha order.
# Note: The request can include any headers; canonical_headers and
# signed_headers lists those that you want to be included in the
# hash of the request. "Host" and "x-amz-date" are always required.
signed_headers = 'host;x-amz-date'
# Step 6: Create payload hash (hash of the request body content). For GET
# requests, the payload is an empty string ("").
payload_hash = hashlib.sha256(('').encode('utf-8')).hexdigest()
# Step 7: Combine elements to create canonical request
canonical_request = method + '\n' + canonical_uri + '\n' + canonical_querystring + '\n'
 + canonical_headers + '\n' + signed_headers + '\n' + payload_hash

# ************* TASK 2: CREATE THE STRING TO SIGN*************
# Match the algorithm to the hashing algorithm you use, either SHA-1 or
# SHA-256 (recommended)
algorithm = 'AWS4-HMAC-SHA256'
credential_scope = datestamp + '/' + region + '/' + service + '/' + 'aws4_request'
string_to_sign = algorithm + '\n' +  amzdate + '\n' +  credential_scope + '\n' +
 hashlib.sha256(canonical_request.encode('utf-8')).hexdigest()
```

```
# ************* TASK 3: CALCULATE THE SIGNATURE *************
# Create the signing key using the function defined above.
signing_key = getSignatureKey(secret_key, datestamp, region, service)
# Sign the string_to_sign using the signing_key
signature = hmac.new(signing_key, (string_to_sign).encode('utf-8'),
 hashlib.sha256).hexdigest()

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

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

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

IDT with FreeRTOS 認定スイート 2.0 (FRQ 2.0)

FreeRTOS 認定スイート 2.0 は FreeRTOS 認定スイートの更新バージョンです。開発者は FRQ 2.0 を使用することをお勧めします。FRQ 2.0 は、FreeRTOS 長期サポート (LTS) ライブラリを実行するデバイスを認定するための関連テストケースで構成されているためです。

IDT for FreeRTOS は、マイクロコントローラー上の FreeRTOS のポートと、それが効果的に通信するかどうかを検証します AWS IoT。具体的には、移植レイヤーインターフェイスが FreeRTOS ライブラリと連動していることを検証し、FreeRTOS テストリポジトリが正しく実装されているかを検証します。また、 を使用してend-to-endのテストも実行します AWS IoT Core。IDT for FreeRTOS で実行されるテストは、FreeRTOS GitHub リポジトリで定義されています。

IDT for FreeRTOS は、テスト対象のマイクロコントローラーデバイス上でフラッシュする組み込みアプリケーションとしてテストを実行します。アプリケーションのバイナリイメージには、FreeRTOS、移植される FreeRTOS インターフェイス、およびボードデバイスドライバーが含まれています。テストの目的は、移植された FreeRTOS インターフェイスがデバイスドライバー上で正しく機能するかどうかを検証することです。

IDT for FreeRTOS は、 AWS Partner Device Catalog にハードウェアを一覧表示 AWS IoT するために に送信できるテストレポートを生成します。詳細については、AWS デバイス認定プログラムを参照してください。

IDT for FreeRTOS は、テスト対象のデバイスに接続されているホストコンピュータ (Windows、macOS、または Linux) 上で動作します。IDT はテストケースの設定とオーケストレーションを行い、結果を集計します。また、テストの実行を管理するためのコマンドラインインターフェイスも用意されています。

デバイスをテストするために、IDT for FreeRTOS は AWS IoT モノ、FreeRTOS グループ、Lambda 関数などのリソースを作成します。これらのリソースを作成するために、IDT for FreeRTOS は で設定された AWS 認証情報`config.json`を使用して、ユーザーに代わって API コールを行います。これらのリソースは、テスト中にさまざまなタイミングでプロビジョニングされます。

IDT for FreeRTOS をホストコンピュータで実行すると、次のステップが実行されます。

1. デバイスおよび認証情報の設定をロードして検証します。
2. 必要なローカルリソースとクラウドリソースを使用して選択したテストを実行します。
3. ローカルリソースとクラウドリソースをクリーンアップします。
4. ボードが資格に必要なテストに合格したかどうかを示すテストレポートを生成します。

LTS 認定の前提条件を設定する

このセクションでは、 でマイクロコントローラーをテストするための前提条件について説明します AWS IoT Device Tester。

FreeRTOS 認定の準備をする

> **Note**
>
> AWS IoT Device Tester for FreeRTOS では、最新の FreeRTOS-LTS バージョンの最新のパッチリリースを使用することを強くお勧めします。

IDT for FRQ 2.0 は FreeRTOS の認定です。IDT FRQ 2.0 を実行して認定を行う前に、「FreeRTOS 認定ガイド」の「Qualifying your board」を完了する必要があります。ライブラリの移植、テスト、および `manifest.yml` のセットアップについては、「FreeRTOS 移植ガイド」の「FreeRTOS ライブラリの移植」を参照してください。FRQ 2.0 には、認定のためのさまざまなプロセスが含まれています。詳細については、「FreeRTOS 認定ガイド」で認定に関する最新の変更内容を参照してください。

IDT を実行するには、FreeRTOS-Libraries-Integration-Tests リポジトリが存在している必要があります。このリポジトリを複製してソースプロジェクトに移植する方法については、「README.md」を参照してください。FreeRTOS-Libraries-Integration-Tests には、IDT を実行するために、プロジェクトのルートに `manifest.yml` が含まれている必要があります。

> **Note**
>
> IDT は、テストリポジトリでの `UNITY_OUTPUT_CHAR` の実装に依存しています。テスト出力ログとデバイスログは、相互にインターリーブしないようにする必要があります。詳細については、「FreeRTOS 移植ガイド」の「ライブラリロギングマクロの実装」セクションを参照してください。

IDT for FreeRTOS のダウンロード

どのバージョンの FreeRTOS にも、認定テストの実行に対応する IDT for FreeRTOS のバージョンがあります。IDT for FreeRTOS の適切なバージョンを AWS IoT Device Tester for FreeRTOS のサポートされているバージョンからダウンロードします。

IDT for FreeRTOS を、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。Microsoft Windows ではパスの長さに文字数の制限があるため、IDT for FreeRTOS は、`C:\` や `D:\` などのルートディレクトリに抽出してください。

> **Note**
>
> NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から複数のユーザーが IDT を実行しないようにする必要があります。そうしないと、クラッシュやデータ破損が発生します。IDT パッケージをローカルドライブに抽出することをお勧めします。

Git のダウンロード

IDT では、ソースコードの整合性を確保するための前提条件として、Git がインストールされている必要があります。

Git をインストールするには、GitHub ガイドの手順を実行します。現在インストールされている Git のバージョンを確認するには、ターミナルで `git --version` コマンドを入力します。

> **Warning**
>
> IDT は Git を使用して、ディレクトリのステータスがクリーンであるかダーティであるかに沿って動作します。Git がインストールされていない場合、`FreeRTOSIntegrity` テストグループは失敗するか、正常に実行されません。IDT が `git executable not found` や `git command not found` などのエラーを返す場合は、Git をインストールまたは再インストールしてから再試行してください。

トピック

- AWS アカウントを作成する
- AWS IoT Device Tester マネージドポリシー
- (オプション) をインストールする AWS Command Line Interface

AWS アカウントを作成する

> **Note**
>
> 完全な IDT 認定スイートは、以下でのみサポートされています。 AWS リージョン
>
> - 米国東部 (バージニア北部)
> - 米国西部 (オレゴン)

- アジアパシフィック (東京)
- 欧州 (アイルランド)

デバイスをテストするために、IDT for FreeRTOS は AWS IoT モノ、FreeRTOS グループ、Lambda 関数などのリソースを作成します。これらのリソースを作成するには、IDT for FreeRTOS では、AWS アカウントと、テストの実行中にユーザーに代わってリソースにアクセスする許可を IDT for FreeRTOS に付与する IAM ポリシーを作成および設定する必要があります。

次の手順では、 AWS アカウントを作成して設定します。

1. アカウントがすでにある場合は AWS 、次のステップに進みます。それ以外の場合は、AWS アカウントを作成します。
2. 「IAM ロールの作成」のステップを実行します。この時点では、アクセス許可やポリシーを追加しないでください。
3. OTA 認定テストを実行するには、ステップ 4 に進みます。それ以外の場合は、ステップ 5 に進みます。
4. OTA IAM アクセス許可インラインポリシーを IAM ロールにアタッチします。

   a.

   > **Important**
   >
   > 次のポリシーテンプレートは、ロールの作成、ポリシーの作成、およびロールへのポリシーのアタッチを行うアクセス許可を IDT に付与します。IDT for FreeRTOS は、ロールを作成するテストに、これらのアクセス許可を使用します。ポリシーテンプレートはユーザーに管理者権限を提供しませんが、アクセス権限を使用して AWS アカウントへの管理者アクセス権を取得できます。

   b. 以下のステップを実行して、必要なアクセス許可を IAM ロールにアタッチします。

      i. [アクセス許可] ページで、[アクセス許可の追加] を選択します。

      ii. [インラインポリシーを作成] を選択します。

      iii. [JSON] タブを選択し、次のアクセス許可を [JSON] テキストボックスにコピーします。中国リージョン以外の場合は、[ほとんどのリージョン] のテンプレートを使用します。中国リージョンの場合は、[北京および寧夏リージョン] のテンプレートを使用します。

Most Regions

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

```
                "iot:Publish",
                "iot:UpdateThingShadow",
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:PutRetentionPolicy"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "iotdeviceadvisor:*",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "logs:DeleteLogGroup",
            "Resource": "arn:aws:logs:*:*:log-group:/aws/iot/
deviceadvisor/*"
        },
        {
            "Effect": "Allow",
            "Action": "logs:GetLogEvents",
            "Resource": "arn:aws:logs:*:*:log-group:/aws/iot/
deviceadvisor/*:log-stream:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreatePolicy",
                "iam:DetachRolePolicy",
                "iam:DeleteRolePolicy",
                "iam:DeletePolicy",
                "iam:CreateRole",
                "iam:DeleteRole",
                "iam:AttachRolePolicy"
            ],
            "Resource": [
                "arn:aws:iam::*:policy/idt*",
                "arn:aws:iam::*:role/idt*"
            ]
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "ssm:GetParameters"
            ],
            "Resource": [
                "arn:aws:ssm:*::parameter/aws/service/ami-amazon-linux-
latest/amzn2-ami-hvm-x86_64-gp2"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:RunInstances",
                "ec2:CreateSecurityGroup",
                "ec2:CreateTags",
                "ec2:DeleteTags"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateKeyPair",
                "ec2:DeleteKeyPair"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:key-pair/idt-ec2-ssh-key-*"
            ]
        },
        {
            "Effect": "Allow",
            "Condition": {
                "StringEqualsIgnoreCase": {
                    "aws:ResourceTag/Owner": "IoTDeviceTester"
                }
            },
            "Action": [
                "ec2:TerminateInstances",
                "ec2:DeleteSecurityGroup",
                "ec2:AuthorizeSecurityGroupIngress",
```

```
                "ec2:RevokeSecurityGroupIngress"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Beijing and Ningxia Regions

次のポリシーテンプレートは、北京および寧夏リージョンで使用できます。

```
{
    "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*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ssm:GetParameters"
            ],
            "Resource": [
                "arn:aws-cn:ssm:*::parameter/aws/service/ami-amazon-
linux-latest/amzn2-ami-hvm-x86_64-gp2"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:RunInstances",
                "ec2:CreateSecurityGroup",
                "ec2:CreateTags",
                "ec2:DeleteTags"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateKeyPair",
                "ec2:DeleteKeyPair"
            ],
            "Resource": [
                "arn:aws-cn:ec2:*:*:key-pair/idt-ec2-ssh-key-*"
            ]
        },
        {
            "Effect": "Allow",
            "Condition": {
                "StringEqualsIgnoreCase": {
                    "aws-cn:ResourceTag/Owner": "IoTDeviceTester"
                }
            },
            "Action": [
                "ec2:TerminateInstances",
                "ec2:DeleteSecurityGroup",
                "ec2:AuthorizeSecurityGroupIngress",
                "ec2:RevokeSecurityGroupIngress"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

iv. 完了したら、[ポリシーの確認] を選択します。

v. ポリシー名として IDTFreeRTOSIAMPermissions と入力します。

vi. [Create policy] を選択します。

5. AWSIoTDeviceTesterForFreeRTOSFullAccess を IAM ロールにアタッチします。

   a. 必要なアクセス許可を IAM ロールにアタッチするには、次の手順を実行します。

      i. [アクセス許可] ページで、[アクセス許可の追加] を選択します。

      ii. [ポリシーのアタッチ] を選択します。

      iii. AWSIoTDeviceTesterForFreeRTOSFullAccess ポリシーを見つけます。チェックボックスをオンにします。

   b. [Add permissions (許可の追加)] を選択します。

6. IDT の認証情報をエクスポートします。詳細については、「Getting IAM role credentials for CLI access」を参照してください。

AWS IoT Device Tester マネージドポリシー

`AWSIoTDeviceTesterForFreeRTOSFullAccess` 管理ポリシーには、バージョンチェック、自動更新機能、メトリクスの収集に関する以下の AWS IoT Device Tester アクセス許可が含まれています。

- `iot-device-tester:SupportedVersion`

  サポートされている製品、テストスイート、IDT バージョンのリストを取得する AWS IoT Device Tester 許可を付与。

- `iot-device-tester:LatestIdt`

  ダウンロード可能な最新の IDT バージョンを取得する AWS IoT Device Tester アクセス許可を付与します。

- `iot-device-tester:CheckVersion`

  IDT、テストスイート、製品のバージョンの互換性をチェックする AWS IoT Device Tester 許可を付与。

- `iot-device-tester:DownloadTestSuite`

  テストスイートの更新をダウンロードする AWS IoT Device Tester 許可を付与。

- `iot-device-tester:SendMetrics`

AWS IoT Device Tester 内部使用状況に関するメトリクスを収集する AWS 許可を付与。

(オプション) をインストールする AWS Command Line Interface

を使用していくつかのオペレーション AWS CLI を実行することをお勧めします。 AWS CLI がインストールされていない場合は、「AWS CLIのインストール」の手順を実行します。

コマンドラインaws configureから を実行して、使用する AWS CLI AWS リージョンの を設定します。IDT for FreeRTOS をサポートする AWS リージョンについては、AWS 「リージョンとエンドポイント」を参照してください。aws configure の詳細については、aws configure を使用したクイック設定を参照してください。

マイクロコントローラーボードの最初のテスト

IDT for FreeRTOS を使用して、FreeRTOS ライブラリの実装をテストできます。ボードのデバイスドライバー用に FreeRTOS ライブラリを移植したら、 AWS IoT Device Tester を使用してマイクロコントローラーボードで認定テストを実行します。

ライブラリ移植レイヤーを追加する

デバイス用に FreeRTOS を移植するには、「FreeRTOS 移植ガイド」を参照してください。FreeRTOS テストリポジトリを実装して FreeRTOS レイヤーを移植する際には、テストリポジトリを含む各ライブラリへのパスを指定した `manifest.yml` を提供する必要があります。このファイルは、ソースコードのルートディレクトリにあります。詳細については、「マニフェストファイルの手順」を参照してください。

クラウドと AWS 通信 AWS IoT Device Tester するように の AWS 認証情報を設定する

クラウドと AWS 通信 AWS IoT Device Tester するには、 の AWS 認証情報を設定する必要があります。詳細については、開発用の AWS 認証情報とリージョンのセットアップを参照してください。有効な AWS 認証情報は*devicetester_extract_location*/devicetester_freertos_*[win|mac|linux]*/configs/config.json設定ファイルで指定されます。

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

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

config.json ファイルの auth 属性には、 AWS 認証を制御するメソッドフィールドがあり、ファイルまたは環境として宣言できます。フィールドを環境に設定すると、ホストマシンの環境変数から AWS 認証情報が取得されます。このフィールドを file に設定すると、指定されたプロファイルが .aws/credentials 設定ファイルからインポートされます。

トピック

- IDT for FreeRTOS でデバイスプールを作成する
- ビルド、フラッシュ、テスト設定を設定する

IDT for FreeRTOS でデバイスプールを作成する

テストするデバイスは、デバイスプールにまとめられます。各デバイスプールは、1 つ以上の同一デバイスで構成されます。IDT for FreeRTOS を設定して、プール内の 1 つのデバイスまたは複数のデバイスをテストすることもがきます。認定プロセスを迅速化するために、IDT for FreeRTOS は、同じ仕様を持つデバイスを並行してテストできます。その際、ラウンドロビンメソッドを使用し、デバイスプール内の各デバイスで異なるテストグループが実行されます。

device.json ファイルの最上位には配列があります。各配列属性は新しいデバイスプールです。各デバイスプールにはデバイス配列属性があり、複数のデバイスが宣言されています。テンプレートには 1 つのデバイスプールがあり、そのデバイスプールにはデバイスが 1 つしかありません。1 つ以上のデバイスをデバイスプールに追加するには、configs フォルダにある device.json テンプレートの devices セクションを編集します。

> **Note**
>
> 同じプール内のすべてのデバイスの技術仕様と SKU は同じでなければなりません。IDT for FreeRTOS は、異なるテストグループに対してソースコードの並列ビルドを可能にするため、ソースコードを IDT for FreeRTOS の抽出されたフォルダにある結果フォルダにコピーします。ビルドコマンドまたはフラッシュコマンドでは、testdata.sourcePath 変数を使用してソースコードパスを参照する必要があります。IDT for FreeRTOS は、この変数を、

コピーしたソースコードの一時パスで置き換えます。詳細については、「IDT for FreeRTOS 変数」を参照してください。

次の例では、device.json ファイルを使用して、複数のデバイスが含まれるデバイスプールを作成しました。

```
[
    {
        "id": "pool-id",
        "sku": "sku",
        "features": [
           {
              "name": "Wifi",
              "value": "Yes | No"
           },
           {
              "name": "Cellular",
              "value": "Yes | No"
           },
           {
              "name": "BLE",
              "value": "Yes | No"
         },
         {
            "name": "PKCS11",
            "value": "RSA | ECC | Both"
         },
         {
             "name": "OTA",
             "value": "Yes | No",
             "configs": [
             {
                 "name": "OTADataPlaneProtocol",
                 "value": "MQTT | HTTP | None"
             }
           ]
         },
         {
            "name": "KeyProvisioning",
            "value": "Onboard | Import | Both | No"
         }
        ],
```

```
        "devices": [
          {
            "id": "device-id",
            "connectivity": {
              "protocol": "uart",
              "serialPort": "/dev/tty*"
            },
            "secureElementConfig" : {
              "publicKeyAsciiHexFilePath": "absolute-path-to/public-key-txt-file:
 contains-the-hex-bytes-public-key-extracted-from-onboard-private-key",
              "publiDeviceCertificateArn": "arn:partition:iot:region:account-
id:resourcetype:resource:qualifier",
              "secureElementSerialNumber": "secure-element-serialNo-value",
              "preProvisioned"           : "Yes | No",
              "pkcs11JITPCodeVerifyRootCertSupport": "Yes | No"
            },
            "identifiers": [
              {
                "name": "serialNo",
                "value": "serialNo-value"
              }
            ]
          }
        ]
    }
]
```

次の属性は、device.json ファイルで使用されます。

id

デバイスのプールを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスは同じタイプであることが必要です。テストスイートを実行するとき、プールのデバイスを使用してワークロードが並列化されます。

sku

テスト対象であるボードを一意に識別する英数字の値。SKU は、適格性が確認されたボードの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog でボードを一覧表示する場合、ここで指定する SKU は、一覧表示プロセスで使用する SKU と一致する必要があります。

`features`

デバイスのサポートされている機能を含む配列。 は、この情報 AWS IoT Device Tester を使用して、実行する認定テストを選択します。

サポートされている値は以下のとおりです。

`Wifi`

ボードが Wi-Fi 機能を備えているかどうかを示します。

`Cellular`

ボードがセルラー機能を備えているかどうかを示します。

`PKCS11`

ボードがサポートする公開鍵暗号化アルゴリズムを指定します。資格認定には PKCS11 が必要です。サポートされている値は、`ECC`、`RSA`、および `Both` です。`Both` は、ボードが ECC と RSA の両方をサポートしていることを示します。

`KeyProvisioning`

信頼された X.509 クライアント証明書をボードに書き込む方法を指定します。

有効な値は、`Import`、`Onboard`、`Both`、および `No` です。認定には、`Onboard`、`Both`、または `No` キープロビジョニングが必要です。`Import` 単独では認定に有効なオプションではありません。

- `Import` は、ボードがプライベートキーのインポートを許可している場合にのみ使用します。`Import` の選択は認定には有効な構成ではないため、テスト目的、特に PKCS11 テストケースでのみ使用してください。`Onboard`、`Both` または `No` は認定に必須です。
- ボードがオンボードプライベートキーをサポートしている場合 (例えば、デバイスに安全な要素がある場合、または独自のデバイスのキーペアと証明書を生成する場合) は、`Onboard` を使用します。各デバイスセクションに `secureElementConfig` 要素を追加し、`publicKeyAsciiHexFilePath` フィールドにパブリックキーファイルへの絶対パスを入力していることを確認します。

- Both は、ボードがキープロビジョニングのためにプライベートキーのインポートとオンボードキーの生成の両方をサポートしている場合に使用します。
- No は、ボードがキープロビジョニングをサポートしていない場合に使用します。No は、デバイスも事前プロビジョニングされている場合のにみ有効なオプションです。

OTA

ボードが無線 (OTA) による更新機能をサポートしているかどうかを示します。OtaDataPlaneProtocol 属性は、デバイスがサポートする OTA データプレーンプロトコルを示します。認定には、HTTP または MQTT データプレーンプロトコルのいずれかを使用する OTA が必要です。テスト中に OTA テストの実行をスキップするには、OTA 機能を No に、OtaDataPlaneProtocol 属性を None に設定します。これは認定の実行にはなりません。

BLE

ボードが Bluetooth Low Energy (BLE) をサポートしているかどうかを示します。

devices.id

テスト対象のデバイスのユーザー定義の一意の識別子。

devices.connectivity.serialPort

テスト対象であるデバイスへの接続に使用される、ホストコンピュータのシリアルポート。

devices.secureElementConfig.PublicKeyAsciiHexFilePath

ボードが pre-provisioned ではない場合、または PublicDeviceCertificateArn が指定されていない場合は必須です。Onboard はキープロビジョニングの必須タイプであるため、現在 FullTransportInterfaceTLS テストグループでは、このフィールドは必須です。デバイスが pre-provisioned の場合、PublicKeyAsciiHexFilePath はオプションであり、含める必要はありません。

次のブロックは、Onboard プライベートキーから抽出された 16 進バイトのパブリックキーを含むファイルへの絶対パスです。

```
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
```

パブリックキーが .der 形式の場合は、パブリックキーを直接 16 進エンコードして 16 進ファイルを生成できます。

.der パブリックキーから 16 進ファイルを生成するには、次の xxd コマンドを入力します。

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

パブリックキーが.pem 形式の場合、base64 でエンコードされたヘッダーとフッターを抽出し、それをバイナリ形式にデコードできます。その後、バイナリ文字列を 16 進エンコードして 16 進ファイルを生成します。

.pem パブリックキーの 16 進ファイルを生成するには、以下を行います。

1. 次の base64 コマンドを実行して、パブリックキーから base64 ヘッダーとフッターを削除します。その後、base64key という名前のデコードされたキーが、ファイル pubkey.der に出力されます。

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

2. 次の xxd コマンドを実行して、pubkey.der を 16 進形式に変換します。結果として生成されたキーは、*outFile* として保存されます。

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

devices.secureElementConfig.PublicDeviceCertificateArn

アップロード先の Secure 要素からの証明書の ARN AWS IoT Core。証明書を にアップロードする方法については AWS IoT Core、「 AWS IoT デベロッパーガイド」の「X.509 クライアント証明書」を参照してください。

devices.secureElementConfig.SecureElementSerialNumber

(オプション) 安全なエレメントのシリアル番号。シリアル番号は、オプションで JITR キープロビジョニング用のデバイス証明書を作成するために使用されます。

devices.secureElementConfig.preProvisioned

(オプション) デバイスにロックダウンされた認証情報を持つ事前プロビジョニングされたセキュア要素があり、オブジェクトをインポート、作成、または破棄できない場合は、「Yes」に設定します。この属性が Yes に設定されている場合は、対応する pkcs11 ラベルを指定する必要があります。

devices.secureElementConfig.pkcs11JITPCodeVerifyRootCertSupport

(オプション) デバイスの corePKCS11 実装が JITP 用のストレージをサポートしている場合は Yes に設定します。これにより、コア PKCS 11 をテストするときに JITP codeverify テストが可能になり、コード検証キー、JITP 証明書、およびルート証明書 PKCS 11 ラベルを指定する必要があります。

identifiers

(オプション) 任意の名前と値のペアの配列。これらの値は、次のセクションで説明されているビルドコマンドやフラッシュコマンドで使用できます。

ビルド、フラッシュ、テスト設定を設定する

IDT for FreeRTOS は、自動的にテストをビルドしてボードにフラッシュします。これを有効にするには、ハードウェアに対してビルドコマンドとフラッシュコマンドを実行するように IDT を設定する必要があります。ビルドとフラッシュのコマンド設定は、config フォルダにある userdata.json テンプレートファイルで設定されています。

デバイスをテストするための設定

ビルド、フラッシュ、およびテストの設定は、configs/userdata.json ファイルで行います。次の JSON の例は、複数のデバイスをテストするために IDT for FreeRTOS を設定する方法を示します。

```
{
    "sourcePath": "</path/to/freertos>",
    "retainModifiedSourceDirectories": true | false,
    "freeRTOSVersion": "<freertos-version>",
    "freeRTOSTestParamConfigPath": "{{testData.sourcePath}}/path/from/source/path/to/
test_param_config.h",
    "freeRTOSTestExecutionConfigPath": "{{testData.sourcePath}}/path/from/source/path/
to/test_execution_config.h",
    "buildTool": {
        "name": "your-build-tool-name",
        "version": "your-build-tool-version",
        "command": [
            "<build command> -any-additional-flags {{testData.sourcePath}}"
        ]
    },
    "flashTool": {
        "name": "your-flash-tool-name",
```

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

            ]
         }
    },
    **********
    This section is used for PKCS #11 labels of private key, public key, device
 certificate, code verification key, JITP certificate, and root certificate.
    When configuring PKCS11, you set up labels and you must provide the labels of the
 device certificate, public key,
    and private key for the key generation type (EC or RSA) it was created with. If
 your device supports PKCS11 storage of JITP certificate,
    code verification key, and root certificate, set
 'pkcs11JITPCodeVerifyRootCertSupport' to 'Yes' in device.json and provide the
 corresponding labels.
    **********
```

```
    "pkcs11LabelConfiguration":{
        "pkcs11LabelDevicePrivateKeyForTLS": "<device-private-key-label>",
        "pkcs11LabelDevicePublicKeyForTLS": "<device-public-key-label>",
        "pkcs11LabelDeviceCertificateForTLS": "<device-certificate-label>",
        "pkcs11LabelPreProvisionedECDevicePrivateKeyForTLS": "<preprovisioned-ec-
device-private-key-label>",
        "pkcs11LabelPreProvisionedECDevicePublicKeyForTLS": "<preprovisioned-ec-device-
public-key-label>",
        "pkcs11LabelPreProvisionedECDeviceCertificateForTLS": "<preprovisioned-ec-
device-certificate-label>",
        "pkcs11LabelPreProvisionedRSADevicePrivateKeyForTLS": "<preprovisioned-rsa-
device-private-key-label>",
        "pkcs11LabelPreProvisionedRSADevicePublicKeyForTLS": "<preprovisioned-rsa-
device-public-key-label>",
        "pkcs11LabelPreProvisionedRSADeviceCertificateForTLS": "<preprovisioned-rsa-
device-certificate-label>",
        "pkcs11LabelCodeVerifyKey": "<code-verification-key-label>",
        "pkcs11LabelJITPCertificate": "<JITP-certificate-label>",
        "pkcs11LabelRootCertificate": "<root-certificate-label>"
     }
  }
```

次のリストは、userdata.json で使用される属性です。

sourcePath

移植された FreeRTOS ソースコードのルートへのパス。

retainModifiedSourceDirectories

(オプション) ビルド時やフラッシュ時に使用した変更後のソースディレクトリをデバッグ用に保持するかどうかをチェックします。true に設定すると、変更されたソースディレクトリは retainedSrc という名前になり、実行された各テストグループの結果ログフォルダに保存されます。含まれない場合、このフィールドはデフォルトで false になります。

freeRTOSTestParamConfigPath

FreeRTOS-Libraries-Integration-Tests 統合用の test_param_config.h ファイルへのパス。このファイルは、ソースコードの root を基準にするために {{testData.sourcePath}} プレースホルダー変数を使用する必要があります。 は、このファイルのパラメータ AWS IoT Device Tester を使用してテストを設定します。

`freeRTOSTestExecutionConfigPath`

FreeRTOS-Libraries-Integration-Tests 統合用の `test_execution_config.h` ファイルへのパス。このファイルは、リポジトリルートを基準にするために `{{testData.sourcePath}}` プレースホルダー変数を使用する必要があります。 は、このファイル AWS IoT Device Tester を使用して、実行するテストを制御します。

`freeRTOSVersion`

実装で使用されているパッチバージョンを含む FreeRTOS のバージョン。AWS IoT Device Tester for FreeRTOS と互換性のある FreeRTOS バージョンについては、「 for FreeRTOS のサポートされているバージョン AWS IoT Device Tester 」を参照してください。 FreeRTOS FreeRTOS

`buildTool`

ソースコードをビルドするコマンド。ビルドコマンドのソースコードパスへの参照はすべて、 AWS IoT Device Tester 変数 に置き換える必要があります`{{testData.sourcePath}}`。`{{config.idtRootPath}}` プレースホルダーを使用して、AWS IoT Device Tester ルートパスを基準にしてビルドスクリプトを参照します。

`flashTool`

イメージをデバイスにフラッシュするコマンド。フラッシュコマンドのソースコードパスへの参照はすべて、 AWS IoT Device Tester 変数 に置き換える必要があります`{{testData.sourcePath}}`。`{{config.idtRootPath}}` プレースホルダーを使用して、AWS IoT Device Tester ルートパスを基準にしたフラッシュスクリプトを参照します。

> **Note**
>
> FRQ 2.0 を使用した新しい統合テスト構造では、`{{enableTests}}` や `{{buildImageName}}` などのパス変数は必要ありません。OTA エンドツーエンドテストは、FreeRTOS-Libraries-Integration-Tests GitHub リポジトリで提供されている設定テンプレートを使用して実行されます。GitHub リポジトリ内のファイルが親ソースプロジェクトに存在する場合、ソースコードはテスト間で変更されません。OTA エンドツーエンド用に別のビルドイメージが必要な場合は、このイメージをビルドスクリプトでビルドし、`otaConfiguration` で指定した `userdata.json` ファイルでそのイメージを指定する必要があります。

testStartDelayms

FreeRTOS test runner がテストの実行を開始する前に待機するミリ秒数を指定します。これは、ネットワークやその他の遅延が原因で、IDT が接続してロギングを開始する前にテスト対象のデバイスが重要なテスト情報の出力を開始した場合に役立ちます。この値は FreeRTOS テストグループにのみ適用され、FreeRTOS test runner を使用しない他のテストグループ (OTA テストなど) には適用されません。[expected 10 but received 5] のようなエラーが表示された場合は、このフィールドを 5000 に設定する必要があります。

echoServerConfiguration

TLS テスト用のエコーサーバーをセットアップするための設定。このフィールドは必須です。

keyGenerationMethod

エコーサーバーはこのオプションで設定されます。オプションは EC または RSA です。

serverPort

エコーサーバーが稼働するポート番号。

otaConfiguration

OTA PAL テストと OTA E2E テストの設定。このフィールドは必須です。

otaE2EFirmwarePath

IDT が OTA エンドツーエンドテストに使用する OTA バイナリイメージへのパス。

otaPALCertificatePath

デバイス上の OTA PAL テストの証明書へのパス。これは署名の検証に使用されます。例えば、ecdsa-sha256-signer.crt.pem などです。

deviceFirmwarePath

起動するファームウェアイメージのハードコード名へのパス。デバイスがファームウェアの起動にファイルシステムを使用しない場合は、このフィールドを `'NA'` として指定します。デバイスがファームウェアブートにファイルシステムを使用する場合は、ファームウェアブートイメージのパスまたは名前を指定します。

codeSigningConfiguration

signingMethod

コード署名の方法。指定できる値は AWS または Custom です。

> **Note**
>
> 北京および寧夏リージョンでは、Custom. AWS code 署名の使用はそのリージョンではサポートされていません。

signerHashingAlgorithm

デバイスでサポートされているハッシュアルゴリズム。想定される値は、SHA1 または SHA256 です。

signerSigningAlgorithm

デバイスでサポートされている署名アルゴリズム。想定される値は、RSA または ECDSA です。

signerCertificate

OTA 用の信頼された証明書。 AWS コード署名方法では、 AWS Certificate Manager にアップロードされた信頼された証明書の Amazon リソースネーム (ARN) を使用します。カスタムコード署名方法では、署名者の証明書ファイルへの絶対パスを使用します。信頼された証明書を作成する方法については、「コード署名証明書の作成」を参照してください。

untrustedSignerCertificate

一部の OTA テストで信頼できない証明書として使用される 2 番目の証明書の ARN またはファイルパス。証明書を作成する方法については、「コード署名証明書の作成」を参照してください。

signerCertificateFileName

デバイスのコード署名証明書のファイル名。この値は、`aws acm import-certificate` コマンド実行時に指定したファイル名と一致する必要があります。

compileSignerCertificate

署名検証証明書のステータスを決定するブール値。有効な値は、`true` および `false` です。

コード署名者の署名検証証明書がプロビジョニングまたはフラッシュされていない場合は、この値を true に設定します。プロジェクトにコンパイルする必要があり

ます。 は信頼できる証明書 AWS IoT Device Tester を取得し、 にコンパイルしますaws_codesigner_certificate.h。

signerPlatform

OTA 更新ジョブの作成時に AWS Code Signer が使用する署名およびハッシュアルゴリズム。現在、このフィールドで可能な値は、AmazonFreeRTOS-TI-CC3220SF と AmazonFreeRTOS-Default です。

- SHA1 および RSA の場合は、AmazonFreeRTOS-TI-CC3220SF を選択します。
- SHA256 および ECDSA の場合は、AmazonFreeRTOS-Default を選択します。
- 設定に SHA256 | RSA または SHA1 | ECDSA が必要な場合は、当社に追加のサポートを依頼してください。
- signingMethod として Custom を選択した場合は、signCommand を設定します。

signCommand

このコマンドには {{inputImageFilePath}} と {{outputSignatureFilePath}} の 2 つのプレースホルダーが必要です。{{inputImageFilePath}} は、IDT によって構築される署名対象のイメージのファイルパスです。{{outputSignatureFilePath}} は、スクリプトによって生成される署名のファイルパスです。

pkcs11LabelConfiguration

PKCS11 ラベル設定には、PKCS11 テストグループを実行するために、デバイス証明書ラベル、公開鍵ラベル、秘密鍵ラベルのラベルセットが少なくとも 1 セット必要です。必要な PKCS11 ラベルは、device.json ファイル内のデバイス設定に基づきます。事前プロビジョニングが device.json で Yes に設定されている場合、必要なラベルは PKCS11 機能に対して選択された内容に応じて、以下のいずれかになります。

- PreProvisionedEC
- PreProvisionedRSA

事前プロビジョニングが device.json で No に設定されている場合、必要なラベルは以下のとおりです。

- pkcs11LabelDevicePrivateKeyForTLS
- pkcs11LabelDevicePublicKeyForTLS
- pkcs11LabelDeviceCertificateForTLS

次の 3 つのラベルは、device.json ファイルで pkcs11JITPCodeVerifyRootCertSupport に対して Yes を選択した場合にのみ必要です。

- pkcs11LabelCodeVerifyKey
- pkcs11LabelRootCertificate
- pkcs11LabelJITPCertificate

これらのフィールドの値は、「FreeRTOS 移植ガイド」で定義されている値と一致する必要があります。

pkcs11LabelDevicePrivateKeyForTLS

(オプション) このラベルはプライベートキーの PKCS #11 ラベルに使用されます。キープロビジョニングのオンボードサポートとインポートサポートがあるデバイスの場合、このラベルがテストに使用されます。このラベルは、事前プロビジョニングされたケースで定義されたラベルとは異なる場合があります。device.json でキープロビジョニングを No に設定し、事前プロビジョニングを Yes に設定した場合、これは未定義になります。

pkcs11LabelDevicePublicKeyForTLS

(オプション) このラベルはパブリックキーの PKCS #11 ラベルに使用されます。キープロビジョニングのオンボードサポートとインポートサポートがあるデバイスの場合、このラベルがテストに使用されます。このラベルは、事前プロビジョニングされたケースで定義されたラベルとは異なる場合があります。device.json でキープロビジョニングを No に設定し、事前プロビジョニングを Yes に設定した場合、これは未定義になります。

pkcs11LabelDeviceCertificateForTLS

(オプション) このラベルはデバイス証明書の PKCS #11 ラベルに使用されます。キープロビジョニングのオンボードサポートとインポートサポートがあるデバイスの場合、このラベルがテストに使用されます。このラベルは、事前プロビジョニングされたケースで定義されたラベルとは異なる場合があります。device.json でキープロビジョニングを No に設定し、事前プロビジョニングを Yes に設定した場合、これは未定義になります。

pkcs11LabelPreProvisionedECDevicePrivateKeyForTLS

(オプション) このラベルはプライベートキーの PKCS #11 ラベルに使用されます。安全な要素またはハードウェアの制限があるデバイスの場合、 AWS IoT 認証情報を保持するためのラベルは異なります。デバイスが EC キーによる事前プロビジョニングをサポートしている場合は、このラベルを指定します。device.json で事前プロビジョニングが Yes に設定されている場合は、このラベルまたは pkcs11LabelPreProvisionedRSADevicePrivateKeyForTLS、あるいはその両方を指定する必要があります。このラベルは、オンボードおよびインポートのケースで定義されたラベルとは異なる場合があります。

pkcs11LabelPreProvisionedECDevicePublicKeyForTLS

(オプション) このラベルはパブリックキーの PKCS #11 ラベルに使用されます。安全な要素またはハードウェアの制限があるデバイスの場合、 AWS IoT 認証情報を保持するためのラベルは異なります。デバイスが EC キーによる事前プロビジョニングをサポートしている場合は、このラベルを指定します。device.json で事前プロビジョニングが Yes に設定されている場合は、このラベルまたは pkcs11LabelPreProvisionedRSADevicePublicKeyForTLS、あるいはその両方を指定する必要があります。このラベルは、オンボードおよびインポートのケースで定義されたラベルとは異なる場合があります。

pkcs11LabelPreProvisionedECDeviceCertificateForTLS

(オプション) このラベルはデバイス証明書の PKCS #11 ラベルに使用されます。安全な要素またはハードウェアの制限があるデバイスの場合、 AWS IoT 認証情報を保持するためのラベルは異なります。デバイスが EC キーによる事前プロビジョニングをサポートしている場合は、このラベルを指定します。device.json で事前プロビジョニングが Yes に設定されている場合は、このラベルまたは pkcs11LabelPreProvisionedRSADeviceCertificateForTLS、あるいはその両方を指定する必要があります。このラベルは、オンボードおよびインポートのケースで定義されたラベルとは異なる場合があります。

pkcs11LabelPreProvisionedRSADevicePrivateKeyForTLS

(オプション) このラベルはプライベートキーの PKCS #11 ラベルに使用されます。安全な要素またはハードウェアの制限があるデバイスの場合、 AWS IoT 認証情報を保持するためのラベルは異なります。デバイスが RSA キーによる事前プロビジョニングをサポートしている場合は、このラベルを指定します。device.json で事前プロビジョニングが Yes に設定されている場合は、このラベルまたは pkcs11LabelPreProvisionedECDevicePrivateKeyForTLS、あるいはその両方を指定する必要があります。

pkcs11LabelPreProvisionedRSADevicePublicKeyForTLS

(オプション) このラベルはパブリックキーの PKCS #11 ラベルに使用されます。安全な要素またはハードウェアの制限があるデバイスの場合、 AWS IoT 認証情報を保持するためのラベルは異なります。デバイスが RSA キーによる事前プロビジョニングをサポートしている場合は、このラベルを指定します。device.json で事前プロビジョニングが Yes に設定されている場合は、このラベルまたは pkcs11LabelPreProvisionedECDevicePublicKeyForTLS、あるいはその両方を指定する必要があります。

pkcs11LabelPreProvisionedRSADeviceCertificateForTLS

(オプション) このラベルはデバイス証明書の PKCS #11 ラベルに使用されます。安全な要素またはハードウェアの制限があるデバイスの場合、 AWS IoT 認証情報を保持するためのラベルは異なります。デバイスが RSA キーによる事前プロビジョニングをサポートしている場合は、このラベルを指定します。`device.json` で事前プロビジョニングが Yes に設定されている場合は、このラベルまたは `pkcs11LabelPreProvisionedECDeviceCertificateForTLS`、あるいはその両方を指定する必要があります。

pkcs11LabelCodeVerifyKey

(オプション) このラベルはコード検証キーの PKCS #11 ラベルに使用されます。デバイスが JITP 証明書、コード検証キー、ルート証明書の PKCS #11 ストレージをサポートしている場合は、このラベルを指定します。`device.json` で `pkcs11JITPCodeVerifyRootCertSupport` が Yes に設定されている場合は、このラベルを指定する必要があります。

pkcs11LabelJITPCertificate

(オプション) このラベルは JITP 証明書の PKCS #11 ラベルに使用されます。デバイスが JITP 証明書、コード検証キー、ルート証明書の PKCS #11 ストレージをサポートしている場合は、このラベルを指定します。`device.json` で `pkcs11JITPCodeVerifyRootCertSupport` が Yes に設定されている場合は、このラベルを指定する必要があります。

IDT for FreeRTOS 変数

コードをビルドしてデバイスをフラッシュするコマンドでは、正常に実行するために接続やデバイスに関するその他の情報が必要になる場合があります。 AWS IoT Device Tester を使用すると、JsonPath を使用してフラッシュコマンドとビルドコマンドでデバイス情報を参照できます。単純な JsonPath 式を使用して、`device.json` ファイルで指定されている情報を取得することができます。

パス変数

IDT for FreeRTOS は、コマンドラインと設定ファイルで使用できる次のようなパス変数を定義します。

{{testData.sourcePath}}

ソースコードパスに拡張されます。この変数を使用する場合は、フラッシュコマンドとビルドコマンドの両方で使用する必要があります。

{{device.connectivity.serialPort}}

シリアルポートに拡張されます。

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

デバイスのシリアル番号に拡張されます。

{{config.idtRootPath}}

AWS IoT Device Tester ルートパスに展開されます。

IDT for FreeRTOS 認定スイート 2.0 (FRQ 2.0) の UI

AWS IoT Device Tester for FreeRTOS (IDT for FreeRTOS) には、IDT コマンドラインアプリケーションおよび関連する設定ファイルを操作できるウェブベースのユーザーインターフェイス (UI) が含まれています。IDT for FreeRTOS UI を使用して、デバイスの新しい設定を作成したり、既存の設定を変更したりします。この UI を使用して、IDT アプリケーションを呼び出し、デバイスに対して FreeRTOS テストを実行することもできます。

コマンドラインを使用して認定テストを実行する方法については、「マイクロコントローラーボードの最初のテスト」を参照してください。

このセクションでは、IDT for FreeRTOS UI の前提条件と、UI から認定テストを実行する方法について説明します。

トピック

- IDT の前提条件を設定する
- IDT UI を使用するように AWS 認証情報を設定する
- IDT for FreeRTOS UI を開く
- 新しい設定を作成する
- 既存の設定を変更する
- 認定テストを実行する

IDT の前提条件を設定する

FreeRTOS UI の AWS IoT Device Tester (IDT) を使用してテストを実行するには、IDT FreeRTOS 認定 (FRQ) 2.x のLTS 認定の前提条件を設定するページの前提条件を満たす必要があります。

IDT UI を使用するように AWS 認証情報を設定する

で作成したユーザーの IAM AWS ユーザー認証情報を設定する必要がありますAWS アカウントを作成する。以下のいずれかの方法で認証情報を指定できます。

- 認証情報ファイルを使用する
- 環境変数を使用する

AWS 認証情報ファイルを使用して認証情報を設定する

IDT では、 AWS CLIと同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

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

次の形式で AWS 認証情報を `credentials` ファイルに追加します。

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

> **Note**
>
> `default` AWS プロファイルを使用しない場合は、IDT for FreeRTOS UI でプロファイル名を指定する必要があります。プロファイルの詳細については、「設定ファイルと認証情報ファイルの設定」を参照してください。

環境変数を使用して AWS 認証情報を設定する

環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッションを閉じると、これらは保存されません。IDT for FreeRTOS UI は、AWS_ACCESS_KEY_IDおよび AWS_SECRET_ACCESS_KEY環境変数を使用して AWS 認証情報を保存します。

これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

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

Windows でこれらの変数を設定するには、set を使用します。

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

IDT for FreeRTOS UI を開く

このトピックでは、IDT for FreeRTOS UI を開いて FreeRTOS 認定スイートを使用する方法について説明します。

IDT for FreeRTOS UI を開く方法

1. サポートされるバージョンの IDT for FreeRTOS をダウンロードします。次に、ダウンロードしたアーカイブを、読み取りおよび書き込みアクセス許可を持っているディレクトリに抽出します。
2. IDT for FreeRTOS のインストールディレクトリに移動します。

   ```
   cd devicetester-extract-location/bin
   ```
3. 次のコマンドを実行して IDT for FreeRTOS UI を開きます。

   Linux

   ```
   .devicetester_ui_linux_x86-64
   ```

   Windows

   ```
   ./devicetester_ui_win_x64-64
   ```

macOS

```
./devicetester_ui_mac_x86-64
```

> **Note**
>
> macOS でシステムが UI を実行できるようにするには、[システム環境設定] -> [セキュリティとプライバシー] に移動します。テストを実行するときは、これをさらに 3 回行う必要がある場合があります。

IDT for FreeRTOS UI がデフォルトのブラウザで開きます。以下のブラウザの最新の 3 つのメジャーバージョンは UI をサポートしています。

- Google Chrome
- Mozilla Firefox
- Microsoft Edge
- MacOS 版 Apple Safari

> **Note**
>
> エクスペリエンスの向上のために、Google Chrome または Mozilla Firefox で IDT for FreeRTOS UI にアクセスすることをお勧めします。Microsoft Internet Explorer は UI ではサポートされていません。

> **Important**
>
> UI を開く前に、 AWS 認証情報を設定する必要があります。認証情報を設定していない場合は、IDT for FreeRTOS UI ブラウザウィンドウを閉じて、「IDT UI を使用するように AWS 認証情報を設定する」の手順に従います。その後に IDT for FreeRTOS UI を再度開きます。

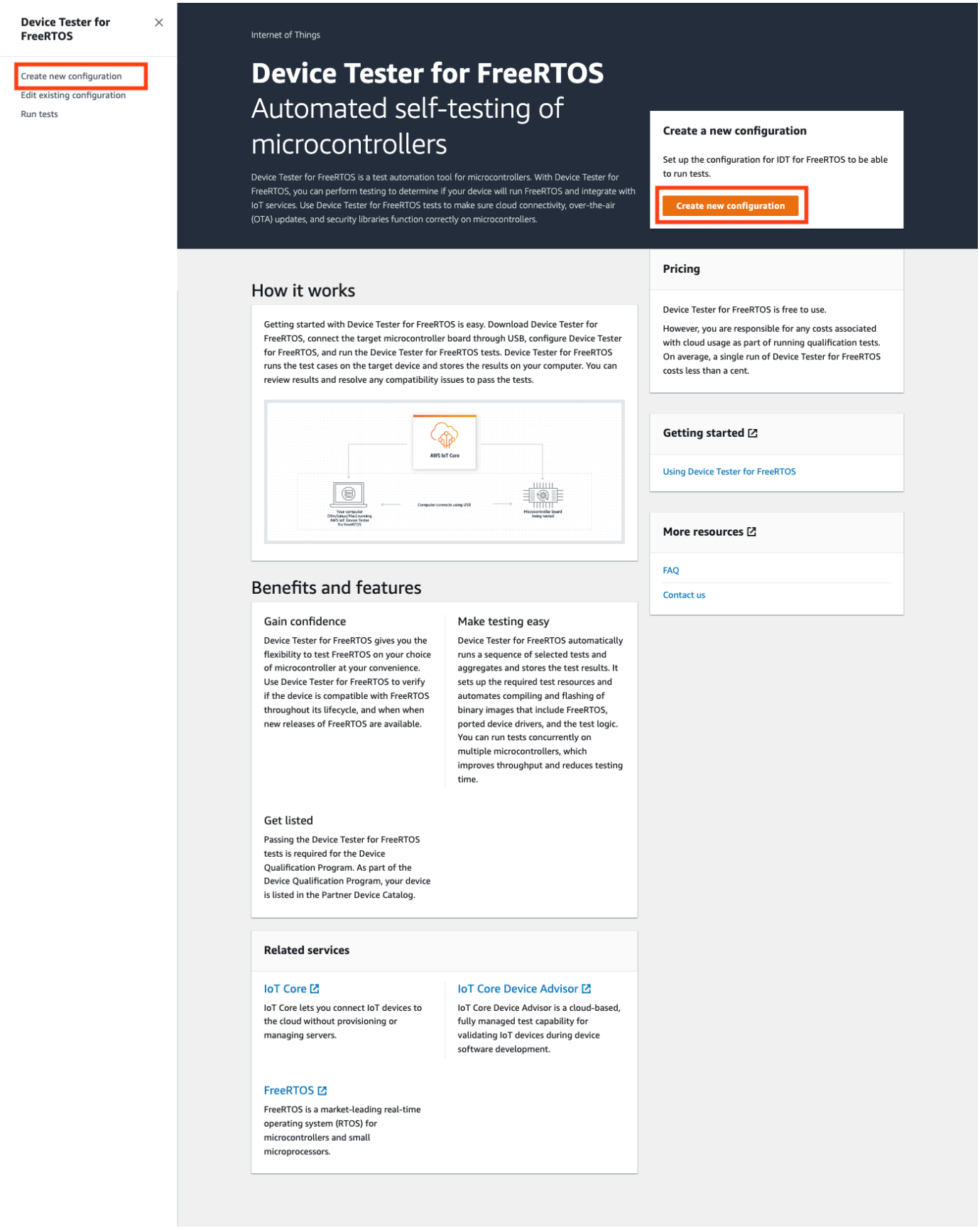
新しい設定を作成する

初めて使用する場合、IDT for FreeRTOS がテストを実行するのに必要な JSON 設定ファイルをセットアップするための新しい設定を作成する必要があります。これにより、テストを実行したり、作成した設定を変更できます。

`config.json`、`device.json`、および `userdata.json` ファイルの例については、マイクロコントローラーボードの最初のテストを参照してください。

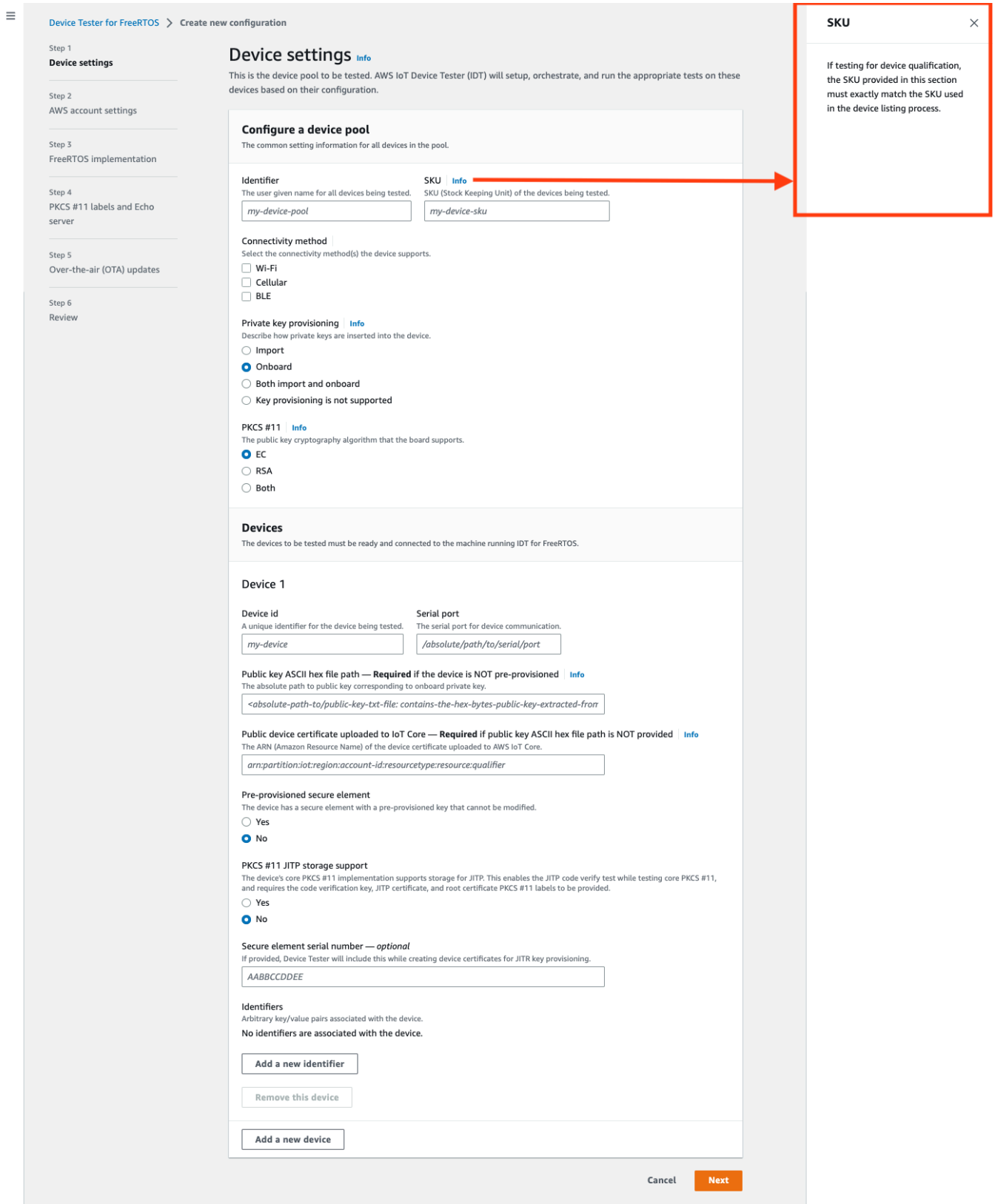
新しい設定を作成するには

1. IDT for FreeRTOS UI でナビゲーションメニューを開いて、[新しい設定を作成] を選択します。

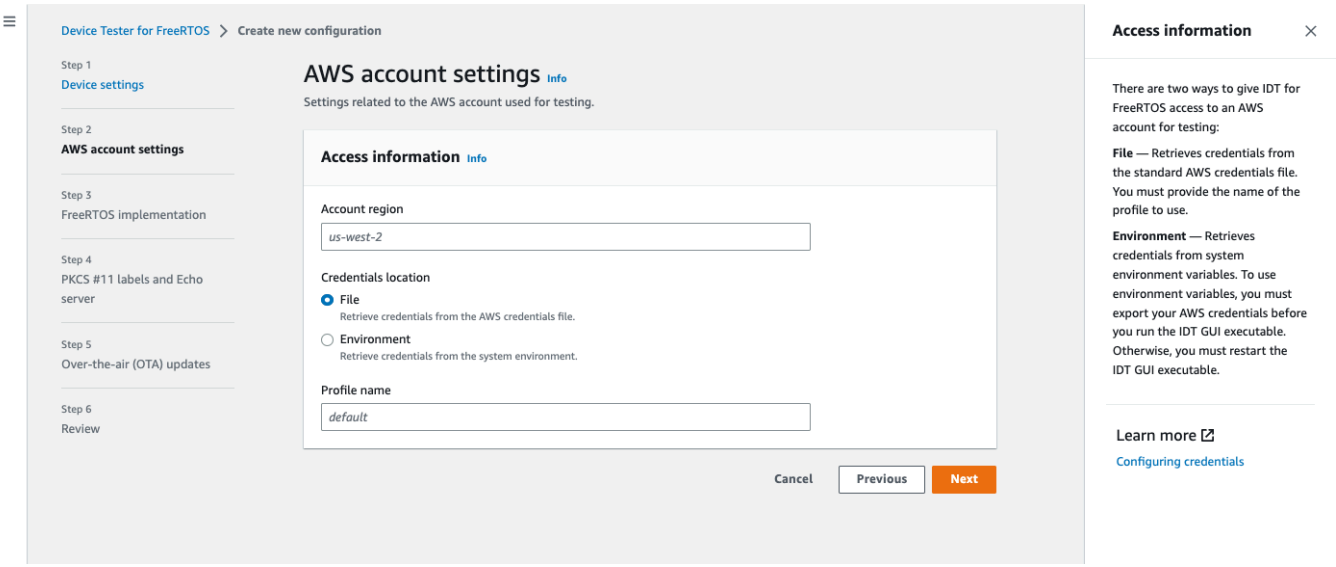

Device Tester for FreeRTOS
Create new configuration
Edit existing configuration
Run tests
Internet of Things
Device Tester for FreeRTOS
Automated self-testing of microcontrollers
Device Tester for FreeRTOS is a test automation tool for microcontrollers. With Device Tester for FreeRTOS, you can perform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

2. 設定ウィザードに従って、認定テストの実行に使用される IDT 構成設定を入力します。このウィザードで *devicetester-extract-location*/config ディレクトリにある JSON 設定ファイル内の次の設定を行います。

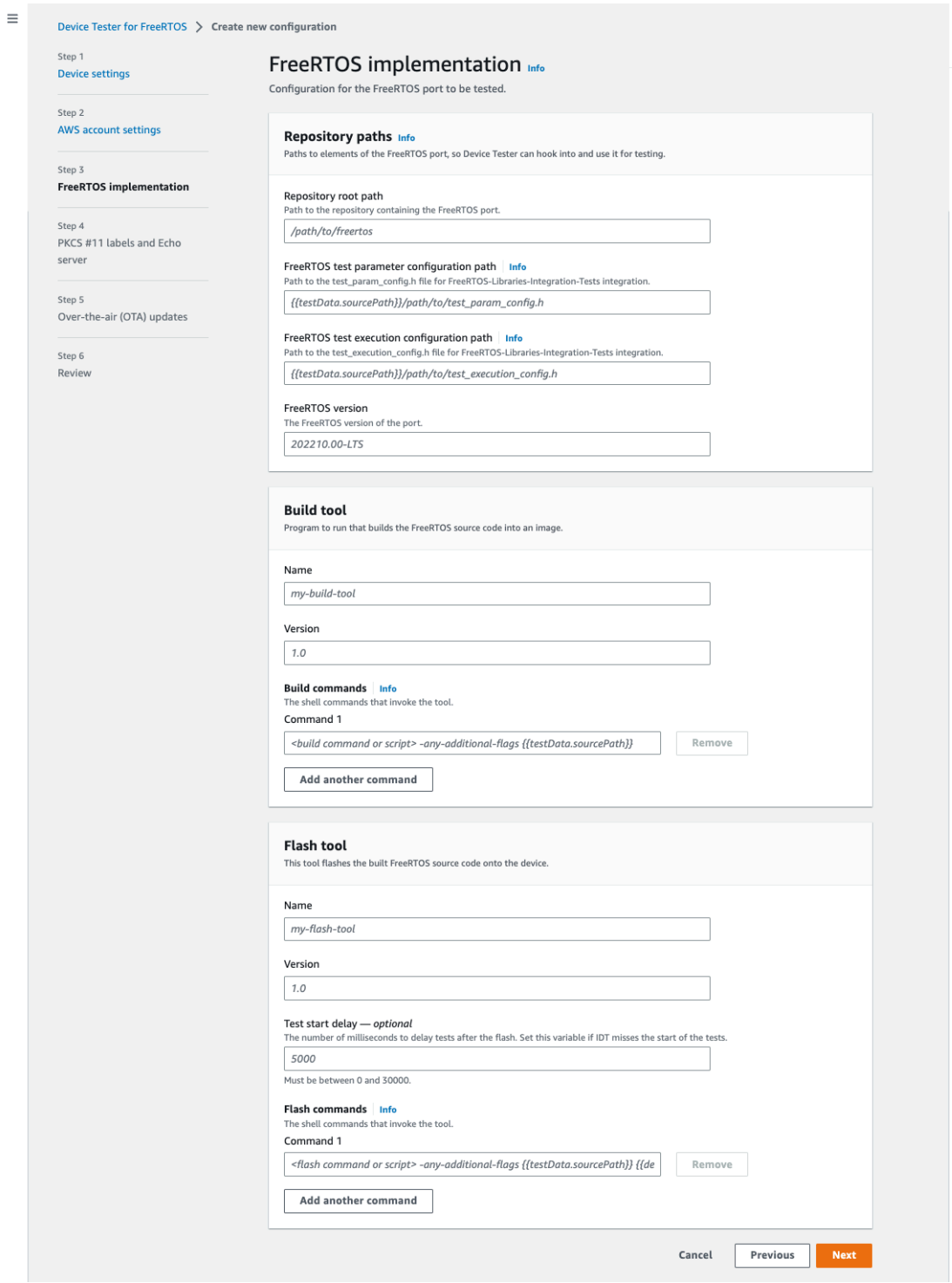
   - [デバイス設定]: テスト対象のデバイス用のデバイスプール設定。これらの設定は、config.json ファイルのデバイスプールに関する id および sku フィールドと [デバイス] ブロックで行われます。



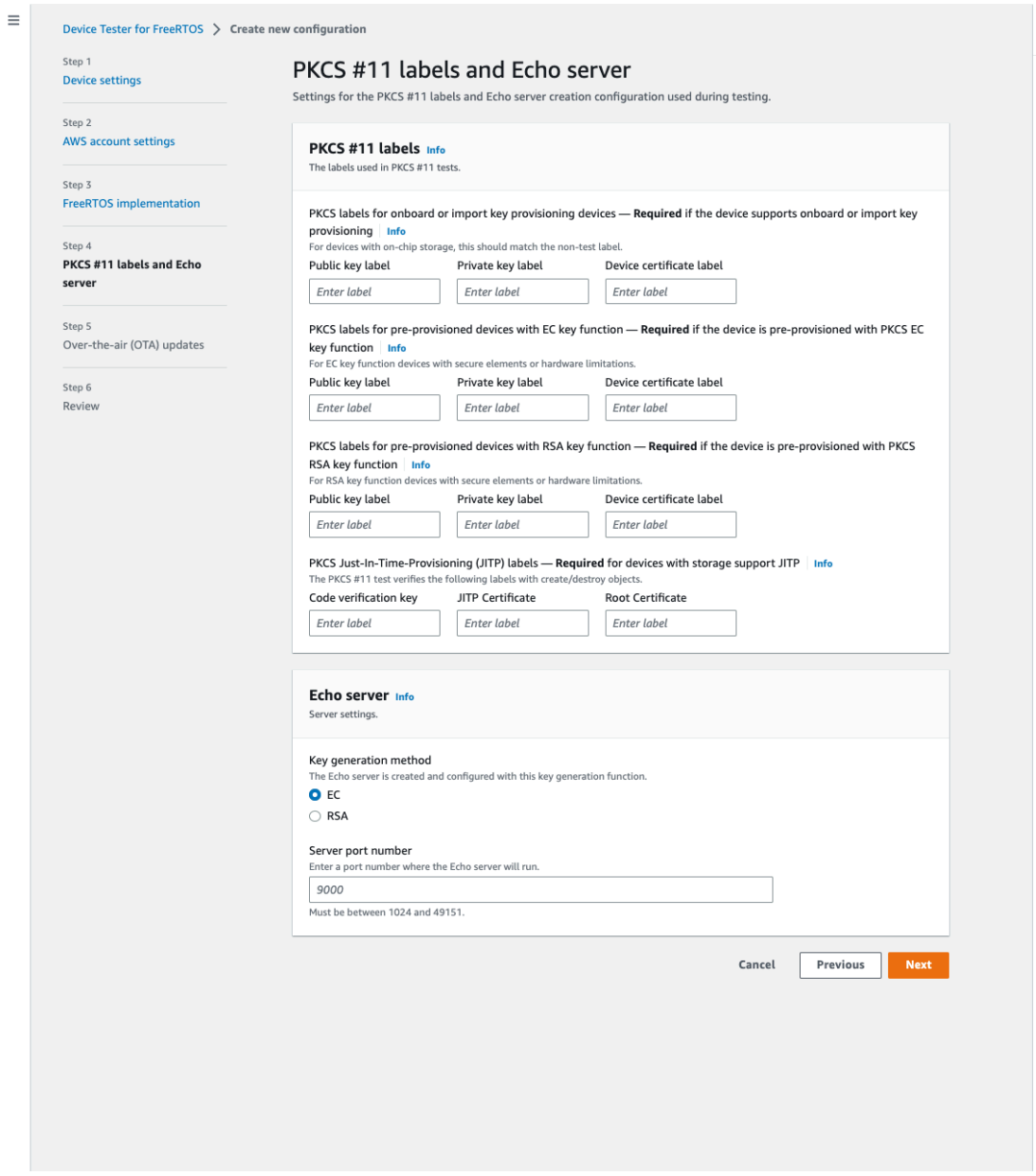
- AWS アカウント設定 – IDT for FreeRTOS AWS アカウント がテスト実行中に AWS リソースを作成するために使用する情報。これらの設定は `config.json` ファイルで行われます。



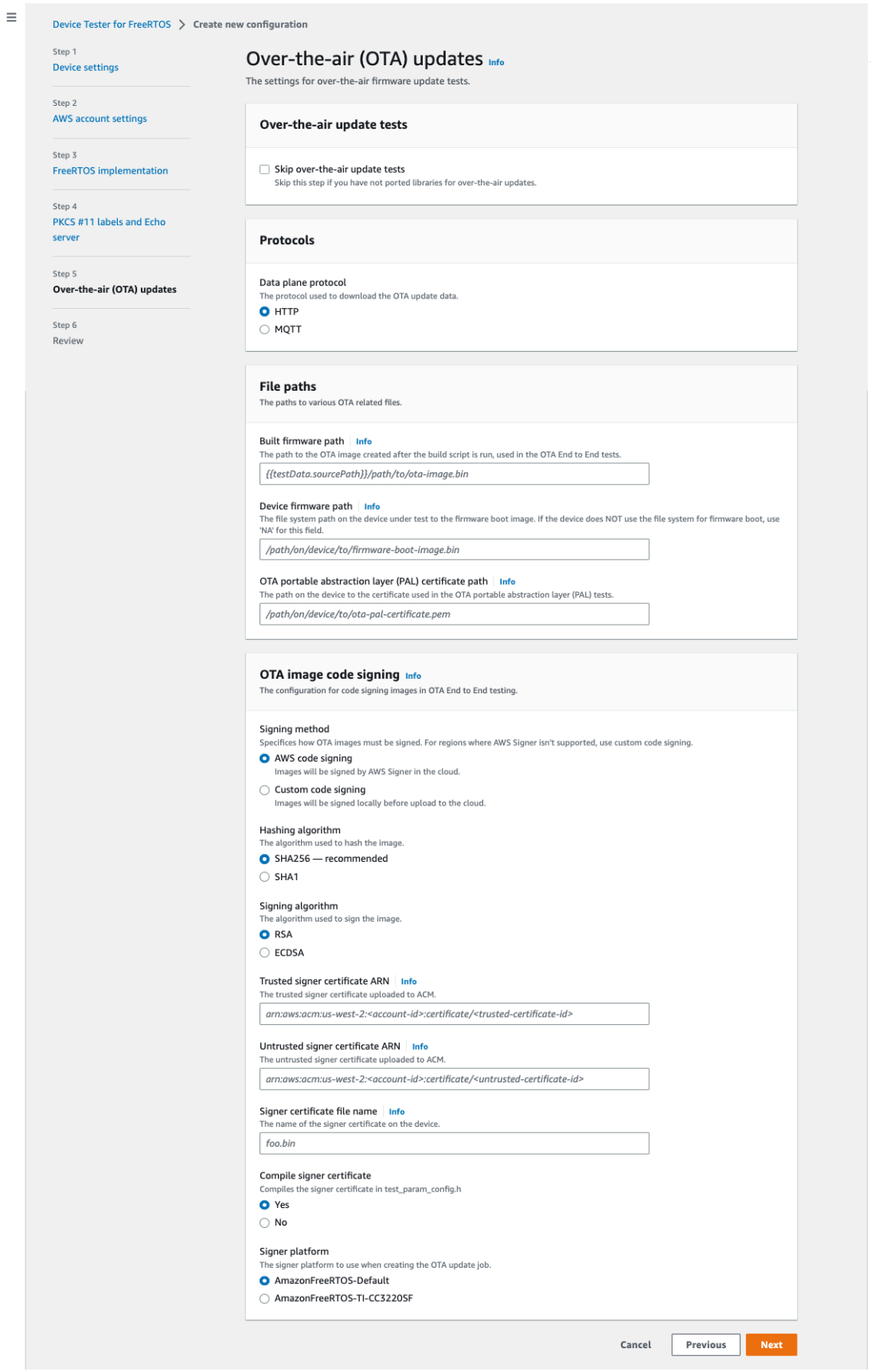
- [FreeRTOS 実装]: FreeRTOS リポジトリと移植されたコードへの絶対パス、および IDT FRQ を実行したい FreeRTOS バージョン。`FreeRTOS-Libraries-Integration-Tests` GitHub リポジトリからの実行およびパラメータ設定ヘッダーファイルへのパス。IDT がボード上でテストを自動的にビルドおよびフラッシュできるようにする、ハードウェア用のビルドコマンドとフラッシュコマンド。これらの設定は `userdata.json` ファイルで行われます。



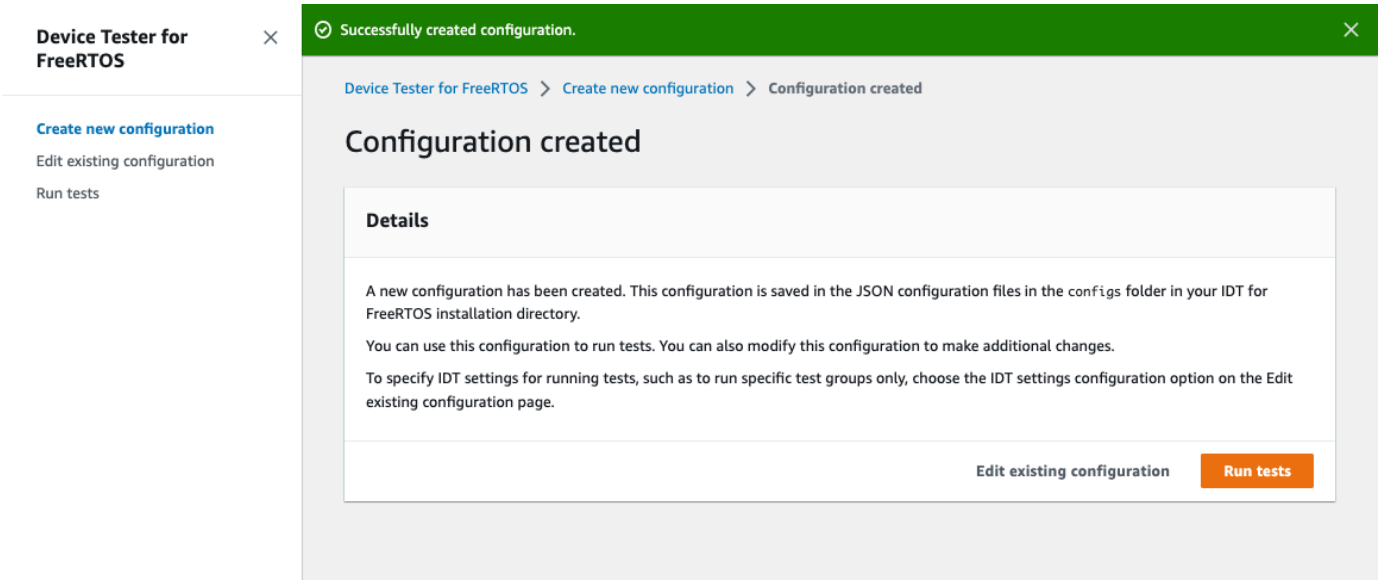
- [PKCS #11 ラベルと Echo サーバー]: キー機能とキープロビジョニング方法に基づいてハードウェアにプロビジョニングされたキーに対応する PKCS #11 ラベル。トランスポートインターフェイステスト用のエコーサーバー構成設定。これらの設定は `userdata.json` および `device.json` ファイルで行われます。



- [OTA (Over-The-Air) 更新] — OTA 機能テストを制御する設定。これらの設定は、`device.json` および `userdata.json` ファイルの `features` ブロックで行われます。


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

3. [Review] (確認) ページで、設定情報を確認します。



設定の確認が完了したら、[Run tests] (テストの実行) を選択して、認定テストを実行します。

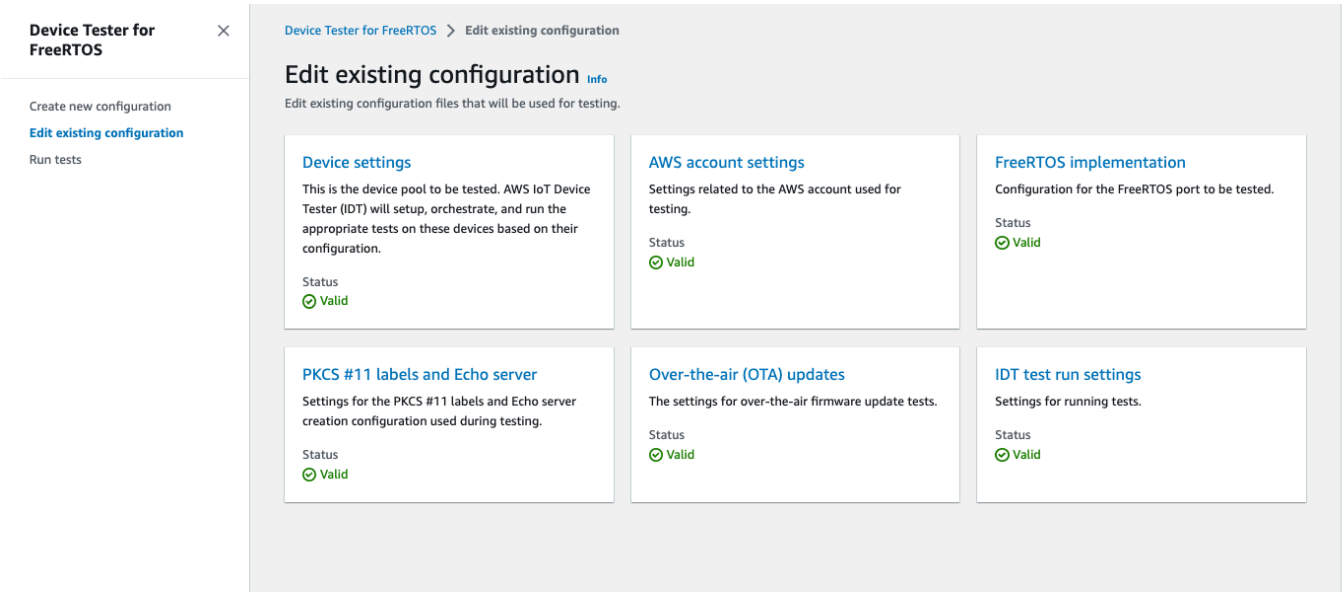
既存の設定を変更する

IDT for FreeRTOS の設定ファイルを既にセットアップしている場合は、IDT for FreeRTOS UI を使用して既存の設定を変更できます。既存の設定ファイルは *devicetester-extract-location*/config ディレクトリにある必要があります。

設定を変更するには

1. IDT for FreeRTOS UI でナビゲーションメニューを開いて、[既存の設定の編集] を選択します。

   設定ダッシュボードには、既存の構成設定に関する情報が表示されます。設定が正しくない、または使用できない場合、設定のステータスは `Error validating configuration` です。

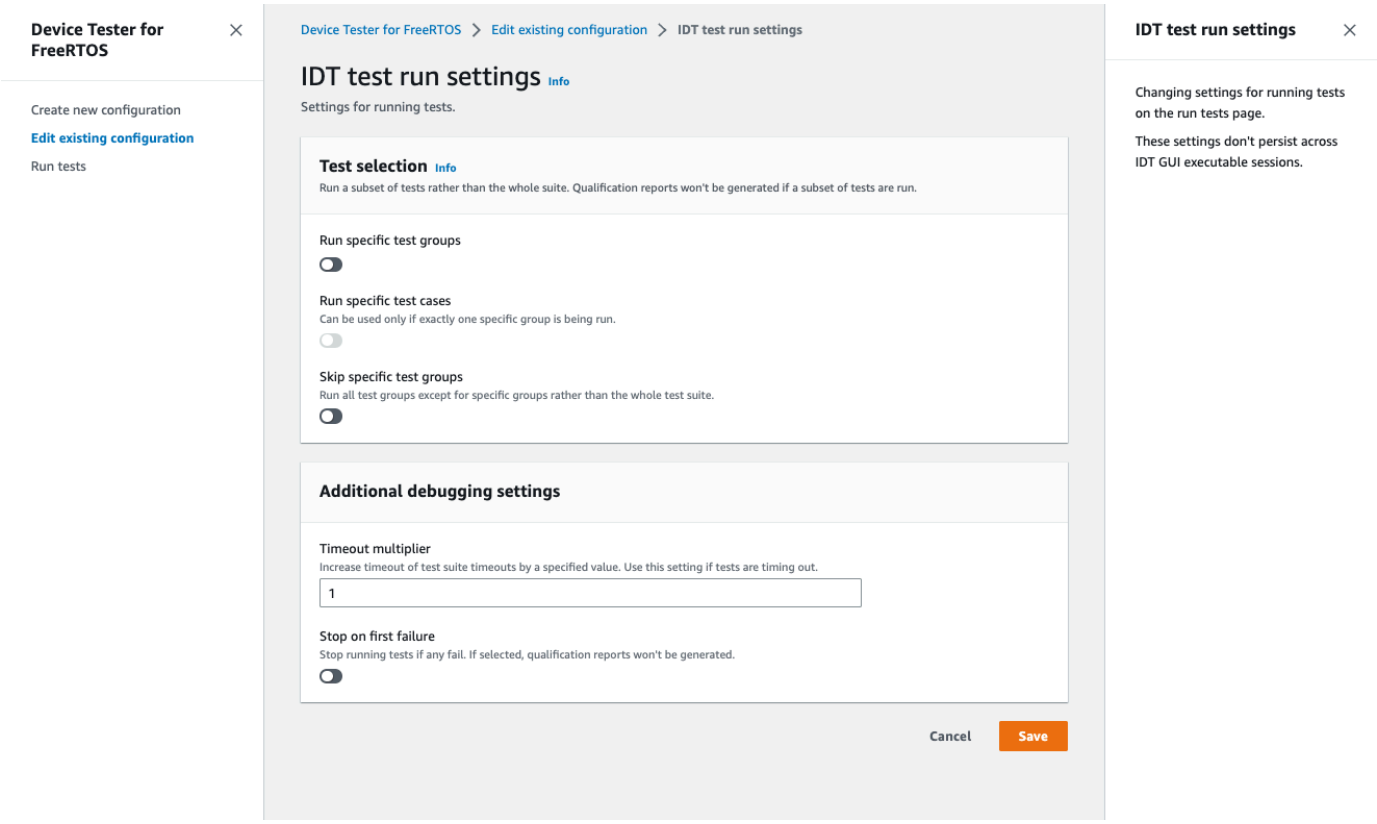


2. 既存の設定を変更するには、以下のステップを実行します。

   a. 構成設定の名前を選択して、設定ページを開きます。

   b. 設定を変更し、[Save] (保存) を選択して、対応する設定ファイルを再生成します。

3. IDT for FreeRTOS のテスト実行設定を変更するには、編集ビューで [IDT テスト実行設定] を選択します。



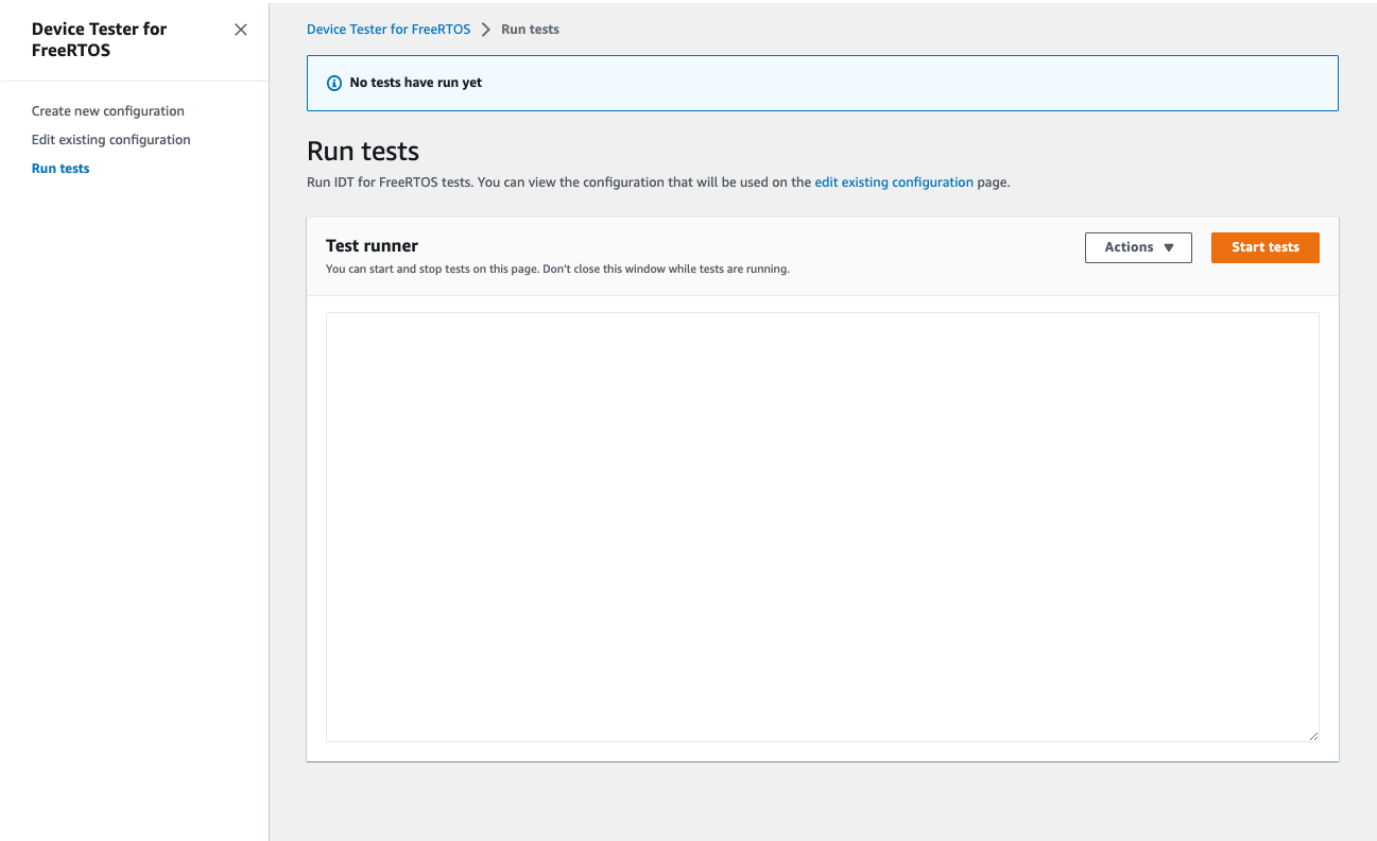
設定の変更が完了したら、すべての構成設定が検証に合格することを確認します。各構成設定のステータスが `Valid` の場合、この設定を使用して認定テストを実行できます。

認定テストを実行する

IDT for FreeRTOS UI の設定を作成したら、認定テストを実行できます。

認定テストを実行するには

1. ナビゲーションメニューで、[Run tests] (テストの実行) を選択します。
2. テストの実行を開始するには、[テストの開始] を選択します。デフォルトでは、該当するすべてのテストがデバイス構成に合わせて実行されます。IDT for FreeRTOS は、すべてのテストが終了すると認定レポートを生成します。

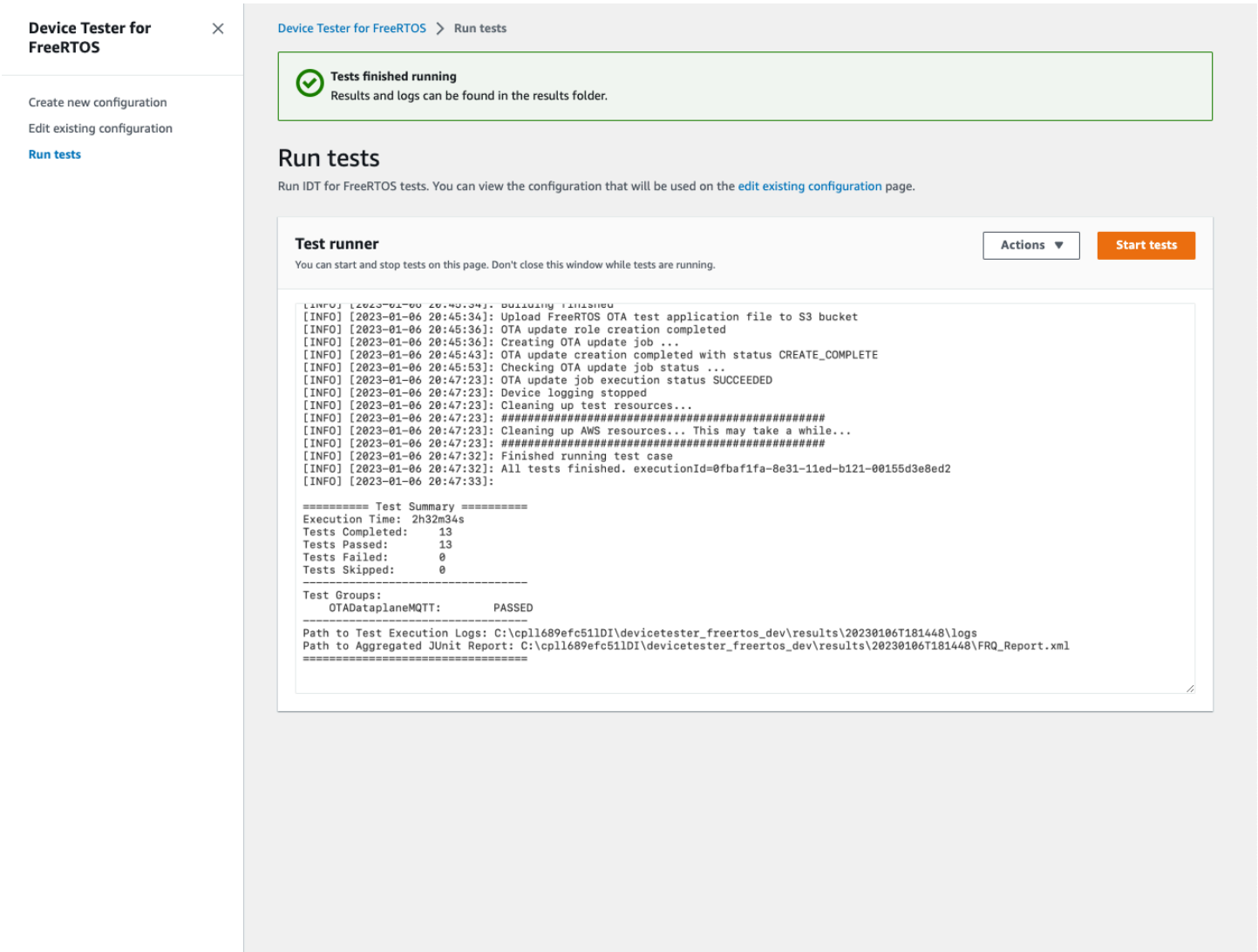


IDT for FreeRTOS が認定テストを実施します。次に、テスト実行の概要とエラーを [テストランナー] コンソールに表示します。テストの実行が完了したら、次の場所でテスト結果とログを確認できます。

- テスト結果は、*devicetester-extract-location*/results/*execution-id* ディレクトリにあります。

- テストのログは *devicetester-extract-location*/results/*execution-id*/logs ディレクトリにあります。

テスト結果とログの詳細については、「」およびIDT for FreeRTOSresultsを表示する「」を参照してくださいIDT for FreeRTOSlogs を表示する。



FreeRTOS 認定 2.0 スイートを実行する

AWS IoT Device Tester for FreeRTOS 実行可能ファイルを使用して IDT for FreeRTOS とやり取りします。以下のコマンドラインの例では、デバイスプール (同一デバイスの集合) に対して適格性確認テストを実行する方法を示します。

IDT v4.5.2 and later

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

デバイスプールに対してテストスイートを実行します。userdata.json ファイルは、*devicetester_extract_location*/devicetester_freertos_*[win|mac|linux]*/configs/ ディレクトリに置く必要があります。

> **Note**
>
> Windows 上で IDT for FreeRTOS を実行する場合、スラッシュ (/) を使用して userdata.json ファイルへのパスを指定します。

特定のテストグループを実行するには、次のコマンドを使用します。

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

1 つのデバイスプールに対して 1 つのスイートを実行する場合 (device.json ファイルで定義したデバイスプールが 1 つしかない場合)、suite-id パラメータと pool-id パラメータは省略可能です。

テストグループの特定のテストケースを実行するには、次のコマンドを使用します。

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

list-test-cases コマンドを使用して、テストグループのテストケースを一覧表示できます。

IDT for FreeRTOS のコマンドラインオプション

group-id

(オプション) 実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。

pool-id

(オプション) テストするデバイスプール。これは、`device.json` で複数のデバイスプールを定義する場合に必要です。デバイスプールが 1 つしかない場合は、このオプションを省略できます。

suite-id

(オプション) 実行するテストスイートのバージョン。指定しない場合、IDT はシステムの tests ディレクトリにある最新バージョンを使用します。

test-id

(オプション) 実行するテスト (カンマ区切りリストとして)。指定した場合、`group-id` は 1 つのグループを指定する必要があります。

Example

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

h

`run-suite` オプションの詳細を確認するには、ヘルプオプションを使用します。

Example

例

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

IDT for FreeRTOS コマンド

IDT for FreeRTOS のコマンドは次のオペレーションをサポートしています。

IDT v4.5.2 and later

help

指定されたコマンドに関する情報を一覧表示します。

list-groups

特定のスイート内のグループを一覧表示します。

list-suites

使用可能なスイートを一覧表示します。

list-supported-products

サポートされている製品とテストスイートのバージョンを一覧表示します。

list-supported-versions

現在の IDT バージョンでサポートされている FreeRTOS およびテストスイートのバージョンを一覧表示します。

list-test-cases

指定したグループのテストケースを一覧表示します。

run-suite

デバイスプールに対してテストスイートを実行します。

`--suite-id` オプションを使用してテストスイートのバージョンを指定するか、そのオプションを省略してシステムの最新バージョンを使用します。

個々のテストケースを実行するには、`--test-id` を使用します。

Example

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

> **Note**
>
> IDT v3.0.0 以降、IDT は新しいテストスイートをオンラインでチェックします。詳細については、「テストスイートのバージョン」を参照してください。

IDT for FreeRTOSresultsを表示する

実行中、IDT はコンソール、ログファイル、テストレポートにエラーを書き込みます。IDT で適格性テストスイートを実行すると、テスト実行の概要がコンソールに書き込まれ、2 つのレポートが生成されます。これらのレポートは *devicetester-extract-location*/results/*execution-id*/ にあります。両レポートでは、適格性確認テストスイートの実行の結果をキャプチャします。

awsiotdevicetester_report.xml は、 AWS Partner Device Catalog にデバイスを一覧表示 AWS するために に送信する認定テストレポートです。レポートには、次の要素が含まれます。

- IDT for FreeRTOS のバージョン。
- テスト済みの FreeRTOS のバージョン。
- 合格したテストに基づいてデバイスでサポートされる FreeRTOS の機能。
- device.json ファイルに記載されている SKU とデバイス名。
- device.json ファイルに記載されているデバイスの機能。
- テストケース結果の要約を集計したもの。
- デバイスの機能に基づいてテストしたライブラリごとのテストケース結果の内訳。

FRQ_Report.xml は、標準 JUnit XML 形式のレポートです。Jenkins、Bamboo など CI/CD プラットフォームに統合することができます。レポートには、次の要素が含まれます。

- テストケース結果の要約を集計したもの。
- デバイスの機能に基づいてテストしたライブラリごとのテストケース結果の内訳。

IDT for FreeRTOS の結果の解釈

awsiotdevicetester_report.xml または FRQ_Report.xml のレポートセクションには、実行したテストの結果が一覧表示されます。

最初の XML タグ <testsuites> は、テストの実行の概要を含みます。例:

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

<testsuites> タグで使用される属性

name

テストスイートの名前。

time

適格性確認スイートの実行所要時間 (秒)。

tests

実行されたテストケースの数。

failures

実行されたテストケースのうち、合格しなかったものの数。

errors

IDT for FreeRTOS が実行できなかったテストケースの数。

disabled

この属性は使用されていないため無視できます。

テストケースに障害やエラーがない場合、デバイスは FreeRTOS を実行するための技術要件を満たし、 AWS IoT サービスと相互運用できます。 AWS Partner Device Catalog にデバイスを一覧表示することを選択した場合は、このレポートを認定の証拠として使用できます。

テストケースに障害やエラーが発生した場合は、<testsuites> XML タグを確認することで、障害の生じたテストケースを特定できます。<testsuites> タグ内の <testsuite> XML タグは、テストグループのテストケース結果の要約を示します。

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

形式は <testsuites> タグと似ていますが、skipped という属性があります。この属性は使用されていないため無視できます。<testsuite> XML タグの内側には、テストグループに対して実行されたそれぞれのテストケースの <testcase> タグがあります。例:

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

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

version

テスト対象の製品のバージョン。

features

検証された機能です。required としてマークされている機能については、資格を得るためにボードを提出するために必要です。次のスニペットは、この情報が awsiotdevicetester_report.xml ファイルで表示される方法を示します。

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

optional としてマークされている機能は、資格を得るために必要ありません。次のスニペットは、オプションの機能を示しています。

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

必要な機能のテストに障害やエラーがない場合、そのデバイスは FreeRTOS を実行するための技術的要件を満たしており、 AWS IoT サービスとの相互運用が可能です。AWS Partner Device Catalog にデバイスを出品する場合は、認定の証拠としてこのレポートを使用できます。

テストに障害やエラーが発生した場合は、<testsuites> XML タグを確認することで、障害の生じたテストを特定できます。<testsuites> タグ内の <testsuite> XML タグは、テストグループのテスト結果の要約を示します。例:

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

形式は <testsuites> タグと似ていますが、使用されていないため無視できる skipped という属性があります。各 <testsuite> XML タグ内には、テストグループの実行されたテスト別の <testcase> タグがあります。例:

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

<testcase> タグで使用される属性

name

テストケースの名前。

attempts

IDT for FreeRTOS がテストケースを実行した回数。

テストに障害やエラーが発生した場合、<failure> タグまたは <error> タグがトラブルシューティングのための情報とともに <testcase> タグに追加されます。例:

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

詳細については、「エラーのトラブルシューティング」を参照してください。

IDT for FreeRTOSlogs を表示する

テストの実行中に IDT for FreeRTOS によって生成されるログは、*devicetester-extract-location*/results/*execution-id*/logs にあります。2 組のログが生成されます。

- test_manager.log

  IDT for FreeRTOS から生成されるログ (設定とレポート生成に関連するログなど) を含みます。
- *test_group_id*/*test_case_id*/*test_case_id*.log

  テスト対象のデバイスからの出力を含む、テストケースのログファイル。ログファイルには、実行されたテストグループとテストケースに従って名前が付けられます。

IDT with FreeRTOS 認定スイート 1.0 (FRQ 1.0)

> **Important**
>
> 2022 年 10 月現在、 AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 は署名付き認定レポートを生成しません。IDT FRQ 1.0 バージョンを使用してデバイスAWS 認定プログラムを通じて、新しい AWS IoT FreeRTOS デバイスを AWS Partner Device

Catalog に一覧表示することはできません。IDT FRQ 1.0 を使用して FreeRTOS デバイスを認定することはできませんが、引き続き FRQ 1.0 を使用して FreeRTOS デバイスをテストすることはできます。FreeRTOS デバイスを認定し、AWS Partner Device Catalog にリストする際には、IDT FRQ 2.0 を使用することが推奨されます。

IDT for FreeRTOS 認定を使用して、FreeRTOS オペレーティングシステムがデバイス上でローカルで動作し、 と通信できることを確認できます AWS IoT。具体的には、FreeRTOS ライブラリの移植レイヤーインターフェイスが正しく実装されていることを検証します。また、 を使用してend-to-endテストも実行します AWS IoT Core。たとえば、ボードで MQTT メッセージを送受信して正しく処理できることを確認します。IDT for FreeRTOS で実行されるテストは、FreeRTOS GitHub リポジトリで定義されています。

テストは、ボードにフラッシュされる埋め込みアプリケーションとして実行されます。アプリケーションのバイナリイメージには、FreeRTOS、半導体ベンダーの移植 FreeRTOS インターフェイス、およびボードデバイスドライバーが含まれています。テストの目的は、移植された FreeRTOS インターフェイスがデバイスドライバーとの組み合わせで正しく機能するかどうかを検証することです。

IDT for FreeRTOS は、 AWS パートナーデバイスカタログにハードウェアを追加 AWS IoT するために に送信できるテストレポートを生成します。詳細については、AWS デバイス認定プログラムを参照してください。

IDT for FreeRTOS は、テスト対象のボードに接続されているホストコンピュータ (Windows、macOS、または Linux) 上で動作します。IDT は、テストケースを実行して結果を集計します。また、テストの実行を管理するためのコマンドラインインターフェイスも用意されています。

IDT for FreeRTOS は、テストデバイスに加えて、認定プロセスを容易にするためにリソース (AWS IoT モノ、FreeRTOS グループ、Lambda 関数など) を作成します。これらのリソースを作成するために、IDT for FreeRTOS は で設定された AWS 認証情報`config.json`を使用してユーザーに代わって API コールを行います。これらのリソースは、テスト中にさまざまなタイミングでプロビジョニングされます。

IDT for FreeRTOS をホストコンピュータで実行すると、次のステップが実行されます。

1. デバイスおよび認証情報の設定をロードして検証します。
2. 必要なローカルリソースとクラウドリソースを使用して選択したテストを実行します。
3. ローカルリソースとクラウドリソースをクリーンアップします。

4. ボードが資格に必要なテストに合格したかどうかを示すテストレポートを生成します。

トピック

- 1.0 認定の前提条件を設定する
- マイクロコントローラーボードの最初のテスト
- IDT ユーザーインターフェイスを使用して FreeRTOS 認定スイートを実行する
- Bluetooth Low Energy テストを実行する
- FreeRTOS 認定スイートを実行する
- IDT for FreeRTOS の結果を表示する
- IDT for FreeRTOS の結果の解釈
- IDT for FreeRTOS ログを表示する

1.0 認定の前提条件を設定する

このセクションでは、 でマイクロコントローラーをテストするための前提条件について説明します AWS IoT Device Tester。

FreeRTOS をダウンロードする

次のコマンドを使用して、GitHub から FreeRTOS のリリースをダウンロードできます。

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

ここで、<FREERTOS_RELEASE_VERSION> は、のサポートされているバージョン AWS IoT Device Testerに記載された IDT バージョンに対応する FreeRTOS のバージョン (202007.00 など) です。これにより、サブモジュールを含むすべてのソースコードを入手でき、FreeRTOS と IDT が相互に対応するバージョンを使用できます。

Windows では、パスの長さは 260 文字に制限されています。FreeRTOS のパス構造は、レベルが多く、深いため、 Windows を使用している場合は、ファイルパスを 260 文字に抑えてください。例えば、`C:\Users\username\programs\projects\myproj\FreeRTOS\` ではなく、`C:\FreeRTOS` に FreeRTOS のクローンを作成します。

LTS ライブラリを使用した FreeRTOS 認定

- マイクロコントローラーを AWS Partner Device Catalog の FreeRTOS の長期サポート (LTS) ベースのバージョンをサポートするように指定するには、マニフェストファイルを指定する必要があります。詳細については、FreeRTOS 資格ガイドの FreeRTOS 資格チェックリストを参照してください。
- マイクロコントローラーが FreeRTOS の LTS ベースのバージョンをサポートしていることを検証し、それを AWS Partner Device Catalog への送信に認定するには、FreeRTOS 認定 AWS IoT Device Tester (FRQ) テストスイートバージョン v1.4.x で (IDT) を使用する必要があります。
- LTS ベースのバージョンの FreeRTOS がサポートされるのは、FreeRTOS のバージョン 202012.xx のみです。

IDT for FreeRTOS のダウンロード

どのバージョンの FreeRTOS にも、認定テストの実行に対応する IDT for FreeRTOS のバージョンがあります。のサポートされているバージョン AWS IoT Device Tester から適切なバージョンの IDT for FreeRTOS をダウンロードします。

IDT for FreeRTOS を、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。Microsoft Windows ではパスの長さに文字数の制限があるため、IDT for FreeRTOS は `C:\` や `D:\` などのルートディレクトリに抽出します。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。このように実行すると、クラッシュまたはデータの破損が発生する可能性があります。IDT パッケージをローカルドライブに抽出することをお勧めします。

AWS アカウントの作成と設定

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。

2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。

2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

AWS IoT Device Tester マネージドポリシー

`AWSIoTDeviceTesterForFreeRTOSFullAccess` 管理ポリシーには、バージョンチェック、自動更新機能、メトリクスの収集に関する以下の AWS IoT Device Tester アクセス許可が含まれています。

- `iot-device-tester:SupportedVersion`

サポートされている製品、テストスイート、IDT バージョンのリストを取得する AWS IoT Device Tester アクセス許可を付与します。

- `iot-device-tester:LatestIdt`

  ダウンロード可能な最新の IDT バージョンを取得する AWS IoT Device Tester アクセス許可を付与します。

- `iot-device-tester:CheckVersion`

  IDT、テストスイート、製品のバージョンの互換性をチェックする AWS IoT Device Tester アクセス許可を付与します。

- `iot-device-tester:DownloadTestSuite`

  テストスイートの更新をダウンロードする AWS IoT Device Tester アクセス許可を付与します。

- `iot-device-tester:SendMetrics`

  AWS IoT Device Tester 内部使用状況に関するメトリクスを収集する AWS アクセス許可を付与します。

(オプション) をインストールする AWS Command Line Interface

一部のオペレーションを実行する AWS CLI には、 を使用することをお勧めします。 AWS CLI がインストールされていない場合は、「AWS CLIのインストール」の手順を実行します。

コマンドラインaws configureから を実行して、使用する AWS CLI AWS リージョンの を設定します。IDT for FreeRTOS をサポートする AWS リージョンについては、AWS 「リージョンとエンドポイント」を参照してください。aws configure の詳細については、aws configure を使用したクイック設定を参照してください。

マイクロコントローラーボードの最初のテスト

FreeRTOS インターフェイスを移植する際、IDT for FreeRTOS を使用してテストできます。ボードのデバイスドライバーに FreeRTOS インターフェイスを移植したら、 AWS IoT Device Tester を使用してマイクロコントローラーボードで認定テストを実行します。

ライブラリ移植レイヤーを追加する

デバイス用に FreeRTOS を移植するには、FreeRTOS 移植ガイドの手順に従います。

AWS 認証情報を設定する

クラウドと AWS 通信 AWS IoT Device Tester するには、 の AWS 認証情報を設定する必要があります。詳細については、開発用の AWS 認証情報とリージョンのセットアップを参照してください。有効な AWS 認証情報は、*devicetester_extract_location*/devicetester_afreertos_*[win|mac|linux]*/configs/config.json設定ファイルで指定する必要があります。

トピック

- IDT for FreeRTOS でデバイスプールを作成する
- ビルド、フラッシュ、テスト設定を設定する

IDT for FreeRTOS でデバイスプールを作成する

テストするデバイスは、デバイスプールにまとめられます。各デバイスプールは、1 つ以上の同一デバイスで構成されます。IDT for FreeRTOS の設定次第で、プール内の 1 つのデバイスをテストすることも、複数のデバイスをテストすることもできます。認定プロセスを迅速化するために、IDT for FreeRTOS は、同じ仕様を持つデバイスを並行してテストできます。その際、ラウンドロビンメソッドを使用し、デバイスプール内の各デバイスで異なるテストグループが実行されます。

1 つ以上のデバイスをデバイスプールに追加するには、configs フォルダにある device.json テンプレートの devices セクションを編集します。

> **Note**
>
> 同じプールにあるデバイスは、すべて同じ技術仕様と同じ SKU を持たなければなりません。

IDT for FreeRTOS は、異なるテストグループに対してソースコードの並列ビルドを可能にするため、ソースコードを IDT for FreeRTOS の抽出されたフォルダにある結果フォルダにコピーします。ビルドコマンドまたはフラッシュコマンドのソースコードパスは、testdata.sourcePath または sdkPath 変数を使用して参照する必要があります。IDT for FreeRTOS は、この変数を、コピーしたソースコードの一時パスで置き換えます。詳細については、IDT for FreeRTOS 変数を参照してください。

次の例では、device.json ファイルを使って複数のデバイスを含んだデバイスプールを作成します。

```
[
    {
        "id": "pool-id",
        "sku": "sku",
        "features": [
            {
                "name": "WIFI",
                "value": "Yes | No"
            },
            {
                "name": "Cellular",
                "value": "Yes | No"
            },
            {
                "name": "OTA",
                "value": "Yes | No",
                "configs": [
                    {
                        "name": "OTADataPlaneProtocol",
                        "value": "HTTP | MQTT"
                    }
                ]
            },
            {
                "name": "BLE",
                "value": "Yes | No"
            },
            {
                "name": "TCP/IP",
                "value": "On-chip | Offloaded | No"
            },
            {
                "name": "TLS",
                "value": "Yes | No"
            },
            {
                "name": "PKCS11",
                "value": "RSA | ECC | Both | No"
            },
            {
                "name": "KeyProvisioning",
                "value": "Import | Onboard | No"
            }
```

```
        ],

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

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

次の属性は、device.json ファイルで使用されます。

id

デバイスのプールを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスは同じタイプであることが必要です。テストスイートを実行するとき、プールのデバイスを使用してワークロードが並列化されます。

sku

テスト対象であるボードを一意に識別する英数字の値。SKU は、適格性が確認されたボードの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog でボードを一覧表示する場合、ここで指定する SKU は、一覧表示プロセスで使用する SKU と一致する必要があります。

`features`

デバイスのサポートされている機能を含む配列。 は、この情報 AWS IoT Device Tester を使用して、実行する認定テストを選択します。

サポートされている値は以下のとおりです。

`TCP/IP`

ボードが TCP/IP スタックをサポートしているかどうか、オンチップ (MCU) でサポートされるのか別のモジュールにオフロードされるのかを示します。資格確認には TCP/IP が必要です。

`WIFI`

ボードが Wi-Fi 機能を備えているかどうかを示します。`Cellular` が `Yes` に設定されている場合は `No` に設定します。

`Cellular`

ボードがセルラー機能を備えているかどうかを示します。`WIFI` が `Yes` に設定されている場合は `No` に設定します。この機能が に設定されている場合Yes、FullSecureSockets テストは AWS t2.micro EC2 インスタンスを使用して実行され、アカウントに追加のコストが発生する可能性があります。詳細については、Amazon EC2 料金表を参照してください。

`TLS`

ボードが TLS をサポートしているかどうかを示します。資格確認には TLS が必要です。

`PKCS11`

ボードがサポートする公開鍵暗号化アルゴリズムを指定します。資格認定には PKCS11 が必要です。サポートされている値は、`ECC`、`RSA`、`Both`、`No` です。`Both` は、ボードが `ECC` と `RSA` の両方のアルゴリズムをサポートしていることを示します。

`KeyProvisioning`

信頼された X.509 クライアント証明書をボードに書き込む方法を指定します。有効な値は、`Import`、`Onboard`、`No` です。資格認定にはキーのプロビジョニングが必要です。

- ボードがプライベートキーのインポートを許可している場合は、`Import` を使用します。IDT は、プライベートキーを作成し、これを FreeRTOS ソースコードに構築します。
- ボードがオンボードプライベートキーの生成をサポートしている場合 (たとえば、デバイスに安全な要素がある場合、または独自のデバイスのキーペアと証明書を生成する場合) は、`Onboard` を使用します。各デバイスセクションに `secureElementConfig` 要素を追加し、`publicKeyAsciiHexFilePath` フィールドにパブリックキーファイルへの絶対パスを入力していることを確認します。
- ボードがキーのプロビジョニングをサポートしていない場合は、`No` を使用します。

`OTA`

ボードが無線 (OTA) による更新機能をサポートしているかどうかを示します。`OtaDataPlaneProtocol` 属性は、デバイスがサポートする OTA データプレーンプロトコルを示します。OTA 機能がデバイスでサポートされていない場合、この属性は無視されます。"Both" を選択すると、MQTT、HTTP の両方と混合テストを実行するため、OTA テストの実行時間が長くなります。

> **Note**
>
> IDT v4.1.0 以降、`OtaDataPlaneProtocol` の値には `HTTP` と `MQTT` のみが使用可能です。

`BLE`

ボードが Bluetooth Low Energy (BLE) をサポートしているかどうかを示します。

`devices.id`

テスト対象のデバイスのユーザー定義の一意の識別子。

`devices.connectivity.protocol`

このデバイスと通信するために使用される通信プロトコル。サポートされている値は `uart` です。

`devices.connectivity.serialPort`

テスト対象であるデバイスへの接続に使用される、ホストコンピュータのシリアルポート。

devices.secureElementConfig.PublicKeyAsciiHexFilePath

オンボードプライベートキーから抽出された 16 進バイトのパブリックキーを含むファイルへの絶対パス。

形式の例:

```
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
```

パブリックキーが .der 形式の場合は、パブリックキーを直接 16 進エンコードして 16 進ファイルを生成できます。

16 進ファイルを生成する .der パブリックキーのコマンド例:

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

パブリックキーが .pem 形式の場合、base64 でエンコードされた部分を抽出してバイナリ形式にデコードし、16 進エンコードして 16 進ファイルを生成できます。

例えば、次のコマンドを使用して、.pem パブリックキーの 16 進ファイルを生成します。

1. キーの base64 でエンコードされた部分を取り出し (ヘッダーとフッターを取り除く)、ファイルに保存し、base64key などの名前を付けて、次のコマンドを使用し .der 形式に変換します。

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

2. xxd コマンドを実行して、16 進形式に変換します。

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

devices.secureElementConfig.SecureElementSerialNumber

(オプション) 安全なエレメントのシリアル番号。FreeRTOS デモ/テストプロジェクトを実行するときにデバイスのパブリックキーと共にシリアル番号が出力される場合、このフィールドを指定します。

devices.secureElementConfig.preProvisioned

(オプション) デバイスにロックダウンされた認証情報を持つ事前プロビジョニングされたセキュア要素があり、オブジェクトをインポート、作成、または破棄できない場合は、「Yes」に設定します。この設定は、`features` で `KeyProvisioning` が「Onboard」に設定され、`PKCS11` が「ECC」に設定されている場合のみ有効です。

identifiers

(オプション) 任意の名前と値のペアの配列。これらの値は、次のセクションで説明されているビルドコマンドやフラッシュコマンドで使用できます。

ビルド、フラッシュ、テスト設定を設定する

IDT for FreeRTOS でテストのビルドとボードへのフラッシュが自動的に実行されるようにするには、ハードウェアでビルドコマンドとフラッシュコマンドを実行するように IDT を設定する必要があります。ビルドとフラッシュのコマンド設定は、`config` フォルダにある `userdata.json` テンプレートファイルで設定されています。

デバイスをテストするための設定

ビルド、フラッシュ、およびテストの設定は、`configs/userdata.json` ファイルで行います。エコーサーバー設定は、`customPath` のクライアントとサーバーの証明書とキーの両方をロードすることによってサポートされます。詳細については、FreeRTOS 移植ガイドのエコーサーバーのセットアップを参照してください。次の JSON の例は、複数のデバイスをテストするために IDT for FreeRTOS を設定する方法を示します。

```
{
    "sourcePath": "/absolute-path-to/freertos",
    "vendorPath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-name",
    // ***********The sdkConfiguration block below is needed if you are not using the
 default, unmodified FreeRTOS repo.
    // In other words, if you are using the default, unmodified FreeRTOS repo then
 remove this block***************
    "sdkConfiguration": {
        "name": "sdk-name",
```

```
        "version": "sdk-version",
        "path": "/absolute-path-to/sdk"
    },
    "buildTool": {
        "name": "your-build-tool-name",
        "version": "your-build-tool-version",
        "command": [
            "{{config.idtRootPath}}/relative-path-to/build-parallel.sh
{{testData.sourcePath}} {{enableTests}}"
        ]
    },
    "flashTool": {
        "name": "your-flash-tool-name",
        "version": "your-flash-tool-version",
        "command": [
            "/{{config.idtRootPath}}/relative-path-to/flash-parallel.sh
{{testData.sourcePath}} {{device.connectivity.serialPort}} {{buildImageName}}"
        ],
        "buildImageInfo" : {
            "testsImageName": "tests-image-name",
            "demosImageName": "demos-image-name"
        }
    },
    "testStartDelayms": 0,
    "clientWifiConfig": {
        "wifiSSID": "ssid",
        "wifiPassword": "password",
        "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA |
eWiFiSecurityWPA2 | eWiFiSecurityWPA3"
    },
    "testWifiConfig": {
        "wifiSSID": "ssid",
        "wifiPassword": "password",
        "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA |
eWiFiSecurityWPA2 | eWiFiSecurityWPA3"
    },
    //**********
    //This section is used to start echo server based on server certificate generation
method,
    //When certificateGenerationMethod is set as Automatic specify the eccCurveFormat
to generate certifcate and key based on curve format,
    //When certificateGenerationMethod is set as Custom specify the certificatePath and
PrivateKeyPath to be used to start echo server
    //**********
```

```
    "echoServerCertificateConfiguration": {
      "certificateGenerationMethod": "Automatic | Custom",
      "customPath": {
          "clientCertificatePath":"/path/to/clientCertificate",
          "clientPrivateKeyPath": "/path/to/clientPrivateKey",
          "serverCertificatePath":"/path/to/serverCertificate",
          "serverPrivateKeyPath": "/path/to/serverPrivateKey"
      },
    "eccCurveFormat": "P224 | P256 | P384 | P521"
    },
    "echoServerConfiguration": {
        "securePortForSecureSocket": 33333, // Secure tcp port used by SecureSocket
 test. Default value is 33333. Ensure that the port configured isn't blocked by the
 firewall or your corporate network
       "insecurePortForSecureSocket": 33334, // Insecure tcp port used by SecureSocket
 test. Default value is 33334. Ensure that the port configured isn't blocked by the
 firewall or your corporate network
       "insecurePortForWiFi": 33335 // Insecure tcp port used by Wi-Fi test. Default
 value is 33335. Ensure that the port configured isn't blocked by the firewall or your
 corporate network
    },
    "otaConfiguration": {
        "otaFirmwareFilePath": "{{testData.sourcePath}}/relative-path-to/ota-image-
generated-in-build-process",
        "deviceFirmwareFileName": "ota-image-name-on-device",
        "otaDemoConfigFilePath": "{{testData.sourcePath}}/relative-path-to/ota-demo-
config-header-file",
        "codeSigningConfiguration": {
            "signingMethod": "AWS | Custom",
            "signerHashingAlgorithm": "SHA1 | SHA256",
            "signerSigningAlgorithm": "RSA | ECDSA",
            "signerCertificate": "arn:partition:service:region:account-
id:resource:qualifier | /absolute-path-to/signer-certificate-file",
            "signerCertificateFileName": "signerCertificate-file-name",
            "compileSignerCertificate": boolean,
            // ***********Use signerPlatform if you choose aws for
 signingMethod***************
            "signerPlatform": "AmazonFreeRTOS-Default | AmazonFreeRTOS-TI-CC3220SF",
            "untrustedSignerCertificate": "arn:partition:service:region:account-
id:resourcetype:resource:qualifier",
            // ***********Use signCommand if you choose custom for
 signingMethod***************
            "signCommand": [
```

```
                "/absolute-path-to/sign.sh {{inputImageFilePath}}
 {{outputSignatureFilePath}}"
            ]
        }
    },
    // ***********Remove the section below if you're not configuring
 CMake***************
    "cmakeConfiguration": {
        "boardName": "board-name",
        "vendorName": "vendor-name",
        "compilerName": "compiler-name",
        "frToolchainPath": "/path/to/freertos/toolchain",
        "cmakeToolchainPath": "/path/to/cmake/toolchain"
    },
    "freertosFileConfiguration": {
        "required": [
            {
                "configName": "pkcs11Config",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_tests/config_files/core_pkcs11_config.h"
            },
            {
                "configName": "pkcs11TestConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_tests/config_files/iot_test_pkcs11_config.h"
            }
        ],
        "optional": [
            {
                "configName": "otaAgentTestsConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_tests/config_files/ota_config.h"
            },
            {
                "configName": "otaAgentDemosConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_demos/config_files/ota_config.h"
            },
            {
                "configName": "otaDemosConfig",
                "filePath": "{{testData.sourcePath}}/vendors/vendor-name/boards/board-
name/aws_demos/config_files/ota_demo_config.h"
            }
        ]
```

```
    }
}
```

次のリストは、userdata.json で使用される属性です。

sourcePath

移植された FreeRTOS ソースコードのルートへのパス。SDK を使用する並行テストの場合、sourcePath は {{userData.sdkConfiguration.path}} プレースホルダーを使用して設定できます。例:

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

vendorPath

ベンダー固有の FreeRTOS コードへのパス。シリアルテストでは、vendorPath を絶対パスとして設定できます。例:

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

並行テストの場合、vendorPath は {{testData.sourcePath}} プレースホルダーを使用して設定できます。例:

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

vendorPath 変数は SDK なしで実行している場合にのみ必要で、それ以外の場合は削除できます。

> **Note**
>
> SDK なしでテストを並行して実行する場合、vendorPath、buildTool、および flashTool フィールドで {{testData.sourcePath}} プレースホルダーを使用する必要があります。1 つのデバイスでテストを実行するときは、vendorPath、buildTool、および flashTool フィールドで絶対パスを使用する必要があります。SDK を使用して実行する場合、{{sdkPath}} プレースホルダーを sourcePath、buildTool、および flashTool コマンドで使用する必要があります。

sdkConfiguration

移植に必要な範囲を超えてファイルやフォルダの構造を変更して FreeRTOS を認定する場合は、このブロックで SDK 情報を設定する必要があります。SDK 内で移植された FreeRTOS を認定しない場合は、このブロックを完全に省略する必要があります。

sdkConfiguration.name

FreeRTOS で使用する SDK の名前。SDK を使用しない場合は、`sdkConfiguration` ブロック全体を省略する必要があります。

sdkConfiguration.version

FreeRTOS で使用する SDK のバージョン。SDK を使用しない場合は、`sdkConfiguration` ブロック全体を省略する必要があります。

sdkConfiguration.path

FreeRTOS コードがある SDK ディレクトリへの絶対パス。SDK を使用しない場合は、`sdkConfiguration` ブロック全体を省略する必要があります。

buildTool

ソースコードをビルドするためのコマンドを含むビルドスクリプト (.bat または .sh) の完全パス。ビルドコマンドのソースコードパスへの参照はすべて AWS IoT Device Tester 変数に置き換え、SDK パスへの`{{testdata.sourcePath}}`参照は に置き換える必要があります`{{sdkPath}}`。IDT の絶対パスまたは相対パスを参照するには、`{{config.idtRootPath}}` プレースホルダーを使用します。

testStartDelayms

FreeRTOS test runner がテストの実行を開始する前に待機するミリ秒数を指定します。これは、ネットワークやその他の遅延が原因でテスト対象のデバイスが重要なテスト情報の出力を開始してから、IDT が接続してロギングを開始する場合に役立ちます。最大許容値は 30000 ms (30 秒) です。この値は FreeRTOS テストグループにのみ適用され、OTA テストなど FreeRTOS test runner を使用しない他のテストグループには適用されません。

flashTool

デバイス用のフラッシュコマンドを含むフラッシュスクリプト (.sh または .bat) の完全パス。フラッシュコマンドのソースコードパスへの参照はすべて IDT for FreeRTOS 変数 `{{testdata.sourcePath}}` に置き換え、SDK パスへ参照はすべて IDT for FreeRTOS 変数 `{{sdkPath}}` に置き換える必要があります。IDT の絶対パスまたは相対パスを参照するには、`{{config.idtRootPath}}` プレースホルダーを使用します。

buildImageInfo

testsImageName

freertos-source/tests フォルダからテストを構築するときにビルドコマンドによって生成されるファイルの名前。

demosImageName

freertos-source/demos フォルダからテストを構築するときにビルドコマンドによって生成されるファイルの名前。

clientWifiConfig

クライアント側の Wi-Fi 設定。Wi-Fi ライブラリテストは、MCU ボードに対し、2 つのアクセスポイントへの接続を要求します。(2 つのアクセスポイントは同じにすることができます)。この属性は、最初のアクセスポイントの Wi-Fi を設定します。一部の Wi-Fi テストケースでは、アクセスポイントにセキュリティが設定されていて、オープンでないことが求められます。両方のアクセスポイントが IDT を実行しているホストコンピュータと同じサブネット上にあることを確認してください。

wifi_ssid

Wi-Fi の SSID。

wifi_password

Wi-Fi のパスワード。

wifiSecurityType

使用されている Wi-Fi セキュリティの種類。次のいずれかの値です。

- eWiFiSecurityOpen
- eWiFiSecurityWEP
- eWiFiSecurityWPA
- eWiFiSecurityWPA2
- eWiFiSecurityWPA3

> **Note**
>
> ボードが Wi-Fi をサポートしていない場合でも、device.json ファイルに clientWifiConfig セクションを含める必要がありますが、属性の値は省略してかまいません。

testWifiConfig

テストの Wi-Fi 設定。Wi-Fi ライブラリテストは、MCU ボードに対し、2 つのアクセスポイントへの接続を要求します。(2 つのアクセスポイントは同じにすることができます)。この属性は、2 番目のアクセスポイントの Wi-Fi を設定します。一部の Wi-Fi テストケースでは、アクセスポイントにセキュリティが設定されていて、オープンでないことが求められます。両方のアクセスポイントが IDT を実行しているホストコンピュータと同じサブネット上にあることを確認してください。

wifiSSID

Wi-Fi の SSID。

wifiPassword

Wi-Fi のパスワード。

wifiSecurityType

使用されている Wi-Fi セキュリティの種類。次のいずれかの値です。

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

> **Note**
>
> ボードが Wi-Fi をサポートしていない場合でも、`device.json` ファイルに `testWifiConfig` セクションを含める必要がありますが、属性の値は省略してかまいません。

echoServerCertificateConfiguration

セキュアソケットテスト用に設定可能なエコーサーバー証明書生成プレースホルダー。このフィールドは必須です。

certificateGenerationMethod

サーバー証明書を自動的に生成するか、手動で生成するかを指定します。

customPath

certificateGenerationMethod が「Custom」の場合は、certificatePath と privateKeyPath が必要です。

certificatePath

サーバー証明書のファイルパスを指定します。

privateKeyPath

プライベートキーのファイルパスを指定します。

eccCurveFormat

ボードがサポートする曲線形式を指定します。PKCS11 が device.json で「ecc」に設定されている場合に必要です。有効な値は「P224」、「P256」、「P384」、または「P521」です。

echoServerConfiguration

WiFi およびセキュアソケットテスト用の設定可能なエコーサーバーポート。このフィールドはオプションです。

securePortForSecureSocket

セキュアソケットテスト用に TLS ありでエコーサーバーをセットアップするために使用されるポート。デフォルト値は 33333 です。設定するポートがファイアウォールまたは社内ネットワークによってブロックされていないことを確認します。

insecurePortForSecureSocket

セキュアソケットテスト用に TLS なしでエコーサーバーをセットアップするために使用されるポート。テストで使用されるデフォルト値は 33334 です。設定するポートがファイアウォールまたは社内ネットワークによってブロックされていないことを確認します。

insecurePortForWiFi

WiFi テスト用に TLS なしでエコーサーバーをセットアップするために使用されるポート。テストで使用されるデフォルト値は 33335 です。設定するポートがファイアウォールまたは社内ネットワークによってブロックされていないことを確認します。

otaConfiguration

OTA 設定。[オプション]

otaFirmwareFilePath

ビルドの後に作成された OTA イメージの完全パス。例えば、{{testData.sourcePath}}/*relative-path/to/ota/image/from/source/root*。

deviceFirmwareFileName

MCU デバイスで、OTA ファームウェアがある場所を示す完全ファイルパス。このフィールドを使用しないデバイスもありますが、値は指定しなければなりません。

otaDemoConfigFilePath

afr-source/vendors/vendor/boards/board/aws_demos/config_files/ 内にある aws_demo_config.h への完全パス。これらのファイルは、FreeRTOS が提供する移植コードテンプレートに含まれています。

codeSigningConfiguration

コード署名の設定。

signingMethod

コード署名の方法。想定される値は、AWS または Custom です。

> **Note**
>
> 北京および寧夏リージョンでは、Custom を使用します。AWS コード署名は、これらのリージョンではサポートされていません。

signerHashingAlgorithm

デバイスでサポートされているハッシュアルゴリズム。想定される値は、SHA1 または SHA256 です。

signerSigningAlgorithm

デバイスでサポートされている署名アルゴリズム。想定される値は、RSA または ECDSA です。

signerCertificate

OTA 用の信頼された証明書。

AWS コード署名方法では、 にアップロードされた信頼された証明書の Amazon リソースネーム (ARN) を使用します AWS Certificate Manager。

カスタムコード署名方法としては、署名者の証明書ファイルへの絶対パスを使用します。

信頼された証明書の作成の詳細については、「コード署名証明書の作成」を参照してください。

signerCertificateFileName

デバイスのコード署名証明書のファイル名。この値は、`aws acm import-certificate` コマンド実行時に指定したファイル名と一致する必要があります。

詳細については、「コード署名証明書の作成」を参照してください。

compileSignerCertificate

`true` コード署名者署名検証証明書がプロビジョニングまたはフラッシュされていない場合、に設定するため、プロジェクトにコンパイルする必要があります。 は信頼できる証明書 AWS IoT Device Tester を取得し、 にコンパイルしますaws_codesigner_certifiate.h。

untrustedSignerCertificate

一部の OTA テストで信頼できない証明書として使用される 2 番目の証明書の ARN またはファイルパス。証明書を作成する方法の詳細については、「コード署名証明書の作成」を参照してください。

signerPlatform

OTA 更新ジョブの作成時に AWS Code Signer が使用する署名およびハッシュアルゴリズム。現在、このフィールドで可能な値は、`AmazonFreeRTOS-TI-CC3220SF` と `AmazonFreeRTOS-Default` です。

- SHA1 および RSA の場合は、`AmazonFreeRTOS-TI-CC3220SF` を選択します。
- SHA256 および ECDSA の場合は、`AmazonFreeRTOS-Default` を選択します。

設定に SHA256 | RSA または SHA1 | ECDSA が必要な場合は、当社に追加のサポートを依頼してください。

`signingMethod` として `Custom` を選択した場合は、`signCommand` を設定します。

signCommand

カスタムコード署名を実行するために使用するコマンド。テンプレートは `/configs/script_templates` ディレクトリにあります。

このコマンドには {{inputImageFilePath}} と {{outputSignatureFilePath}} の 2 つのプレースホルダーが必要です。{{inputImageFilePath}} は、IDT によって構築される署名対象のイメージのファイルパスです。{{outputSignatureFilePath}} は、スクリプトによって生成される署名のファイルパスです。

cmakeConfiguration

CMake の設定 [オプション]

> **Note**
>
> CMake テストケースを実行するには、ベンダー名、ボード名、および frToolchainPath または compilerName を指定する必要があります。CMake ツールチェーンへのカスタムパスがある場合は、cmakeToolchainPath を指定することもできます。

boardName

テスト対象ボードの名前。ボード名は、*path/to/afr/source/code*/vendors/*vendor*/boards/*board* のフォルダ名と同じにする必要があります。

vendorName

テスト対象ボードのベンダー名。ベンダー名は、*path/to/afr/source/code*/vendors/*vendor* のフォルダ名と同じにする必要があります。

compilerName

コンパイラ名。

frToolchainPath

コンパイラツールチェーンへの完全修飾パス。

cmakeToolchainPath

CMake ツールチェーンへの完全修飾パス。このフィールドはオプションです

freertosFileConfiguration

IDT がテストを実行する前に変更する FreeRTOS ファイルの設定。

required

このセクションでは、設定ファイルを移動済みの必須テストを指定します (PKCS11、TLS など)。

configName

設定対象のテストの名前。

filePath

freertos リポジトリ内の設定ファイルの絶対パス。{{testData.sourcePath}} 変数を使用してパスを定義します。

optional

このセクションでは、設定ファイルを移動済みの必須テストを指定します (OTA、WiFi など)。

configName

設定対象のテストの名前。

filePath

freertos リポジトリ内の設定ファイルの絶対パス。{{testData.sourcePath}} 変数を使用してパスを定義します。

> **Note**
>
> CMake テストケースを実行するには、ベンダー名、ボード名、および afrToolchainPath または compilerName を指定する必要があります。CMake ツールチェーンへのカスタムパスがある場合は、cmakeToolchainPath を指定することもできます。

IDT for FreeRTOS 変数

コードを構築してデバイスをフラッシュするコマンドでは、正常に実行するために接続やデバイスに関するその他の情報が必要になる場合があります。 AWS IoT Device Tester では、JsonPath を使用してフラッシュおよびビルドコマンドでデバイス情報を参照できます。単純な JsonPath 式を使用して、device.json ファイルで指定されている情報を取得することができます。

パス変数

IDT for FreeRTOS は、コマンドラインと設定ファイルで使用できる次のようなパス変数を定義します。

{{testData.sourcePath}}

ソースコードパスに拡張されます。この変数を使用する場合は、フラッシュコマンドとビルドコマンドの両方で使用する必要があります。

{{sdkPath}}

ビルドコマンドとフラッシュコマンドで使用する場合、`userData.sdkConfiguration.path` で値に拡張されます。

{{device.connectivity.serialPort}}

シリアルポートに拡張されます。

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

デバイスのシリアル番号に拡張されます。

{{enableTests}}

ビルドがテスト用 (値 1) であるか、デモ (値 0) であるかを示す整数値。

{{buildImageName}}

ビルドコマンドによってビルドされたイメージのファイル名。

{{otaCodeSignerPemFile}}

OTA Code Signer の PEM ファイル。

{{config.idtRootPath}}

AWS IoT Device Tester ルートパスに展開します。この変数は、ビルドコマンドとフラッシュコマンドで使用される場合、IDT の絶対パスを置き換えます。

IDT ユーザーインターフェイスを使用して FreeRTOS 認定スイートを実行する

IDT v4.3.0 以降、 AWS IoT Device Tester for FreeRTOS (IDT-FreeRTOS) には、IDT コマンドライン実行可能ファイルと関連する設定ファイルを操作できるウェブベースのユーザーインターフェイス

が含まれています。IDT-FreeRTOS UI を使用して新しい設定を作成すると、IDT テストの実行や既存の設定の変更ができます。UI を使用して IDT 実行可能ファイルを呼び出してテストを実行することもできます。

IDT-FreeRTOS UI には次の機能があります。

- IDT-FreeRTOS テストの設定ファイルを簡単にセットアップできます。
- IDT-FreeRTOS を使用して認定テストを簡単に実行できます。

コマンドラインを使用して認定テストを実行する方法については、マイクロコントローラーボードの最初のテストを参照してください。

このセクションでは、IDT-FreeRTOS UI を使用するための前提条件と、UI で認定テストの実行を開始する方法について説明します。

トピック

- FreeRTOS 認定スイートを実行するための前提条件を設定する
- IDT-FreeRTOS UI の使用を開始する

FreeRTOS 認定スイートを実行するための前提条件を設定する

このセクションでは、 AWS IoT Device Testerでマイクロコントローラーをテストするための前提条件について説明します。

トピック

- サポートされているウェブブラウザを使用する
- FreeRTOS をダウンロードする
- IDT for FreeRTOS のダウンロード
- AWS アカウントの作成と設定
- AWS IoT Device Tester マネージドポリシー

サポートされているウェブブラウザを使用する

IDT-FreeRTOS UI では以下のウェブブラウザがサポートされています。

ブラウザ	バージョン
Google Chrome	最新の 3 つのメジャーバージョン
Mozilla Firefox	最新の 3 つのメジャーバージョン
Microsoft Edge	最新の 3 つのメジャーバージョン
MacOS 版 Apple Safari	最新の 3 つのメジャーバージョン

より良いエクスペリエンスを得るには、Google Chrome または Mozilla Firefox を使用することをお勧めします。

> **Note**
>
> IDT-FreeRTOS UI は Microsoft Internet Explorer をサポートしていません。

FreeRTOS をダウンロードする

次のコマンドを使用して、GitHub から FreeRTOS のリリースをダウンロードできます。

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

ここで、<FREERTOS_RELEASE_VERSION> は、のサポートされているバージョン AWS IoT Device Testerに記載された IDT バージョンに対応する FreeRTOS のバージョン (202007.00 など) です。これにより、サブモジュールを含むすべてのソースコードを入手でき、FreeRTOS と IDT が相互に対応するバージョンを使用できます。

Windows では、パスの長さは 260 文字に制限されています。FreeRTOS のパスの構造は、レベルが多く、深いため、 Windows を使用している場合は、ファイルパスを 260 文字に抑えてください。例えば、`C:\Users\username\programs\projects\myproj\FreeRTOS\` ではなく、`C:\FreeRTOS` に FreeRTOS のクローンを作成します。

LTS 認定 (LTS ライブラリを使用する FreeRTOS の認定) に関する考慮事項

- マイクロコントローラーを AWS Partner Device Catalog の FreeRTOS の長期サポート (LTS) ベースのバージョンをサポートするように指定するには、マニフェストファイルを指定する必要があります。詳細については、FreeRTOS 資格ガイドの FreeRTOS 資格チェックリストを参照してください。
- マイクロコントローラーが FreeRTOS の LTS ベースのバージョンをサポートし、 AWS それを Partner Device Catalog に送信するための認定を受けるには、 AWS IoT Device Tester FreeRTOS 認定 (FRQ) テストスイートバージョン v1.4.x で (IDT) を使用する必要があります。
- LTS ベースのバージョンの FreeRTOS がサポートされるのは、FreeRTOS のバージョン 202012.xx のみです。

IDT for FreeRTOS のダウンロード

どのバージョンの FreeRTOS にも、認定テストの実行に対応する IDT for FreeRTOS の対応バージョンがあります。のサポートされているバージョン AWS IoT Device Tester から適切なバージョンの IDT for FreeRTOS をダウンロードします。

IDT for FreeRTOS を、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。Microsoft Windows ではパスの長さに文字数の制限があるため、IDT for FreeRTOS は `C:\` や `D:\` などのルートディレクトリに抽出します。

> **Note**
>
> IDT パッケージをローカルドライブに抽出することをお勧めします。複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行できるようにすると、システムが応答しないか、データが破損する可能性があります。

AWS アカウントの作成と設定

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

AWS IoT Device Tester マネージドポリシー

`AWSIoTDeviceTesterForFreeRTOSFullAccess` マネージドポリシーには、デバイステスターがメトリクスを実行および収集できるように、次のアクセス許可が含まれています。

- `iot-device-tester:SupportedVersion`

  IDT でサポートされている FreeRTOS のバージョンとテストスイートのバージョンのリストを取得するアクセス許可を付与し、それらのバージョンを AWS CLIから使用できるようにします。
- `iot-device-tester:LatestIdt`

ダウンロード可能な AWS IoT Device Tester 最新バージョンを取得するアクセス許可を付与します。

- `iot-device-tester:CheckVersion`

  製品、テストスイート、および AWS IoT Device Tester バージョンの組み合わせに互換性があることを確認するアクセス許可を付与します。

- `iot-device-tester:DownloadTestSuite`

  テストスイートをダウンロードするアクセス許可を付与 AWS IoT Device Tester します。

- `iot-device-tester:SendMetrics`

  AWS IoT Device Tester 使用状況メトリクスデータを公開するアクセス許可を付与します。

IDT-FreeRTOS UI の使用を開始する

このセクションでは、IDT-FreeRTOS UI を使用して設定を作成または変更する方法と、テストの実行方法について説明します。

トピック

- AWS 認証情報を設定する
- IDT-FreeRTOS UI を開く
- 新しい設定を作成する
- 既存の設定を変更する
- 認定テストを実行する

AWS 認証情報を設定する

で作成した AWS ユーザーの認証情報を設定する必要がありますAWS アカウントの作成と設定。以下のいずれかの方法で認証情報を指定できます。

- 認証情報ファイルを使用する
- 環境変数を使用する

AWS 認証情報ファイルを使用して認証情報を設定する

IDT では、 AWS CLIと同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

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

次の形式で AWS 認証情報を `credentials` ファイルに追加します。

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

> **Note**
>
> `default` AWS プロファイルを使用しない場合は、IDT-FreeRTOS UI でプロファイル名を必ず指定してください。プロファイルの詳細については、「設定と認証情報ファイルの設定」を参照してください。

環境変数を使用して AWS 認証情報を設定する

環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッションを閉じると、これらは保存されません。IDT-FreeRTOS UI は、環境変数の `AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY` を使用して AWS 認証情報を保存します。

これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

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

Windows でこれらの変数を設定するには、set を使用します。

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

IDT-FreeRTOS UI を開く

IDT-FreeRTOS UI を開くには

1. サポートされているバージョンの IDT-FreeRTOS をダウンロードし、ダウンロードしたアーカイブを、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。
2. 次のコマンドを実行して IDT-FreeRTOS インストールディレクトリに移動します。

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

3. 次のコマンドを実行して IDT-FreeRTOS UI を開きます。

   Linux

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

   Windows

   ```
   ./devicetestergui_win_x64-64
   ```

   macOS

   ```
   ./devicetestergui_mac_x86-64
   ```

   > **Note**
   >
   > Mac では、システムで UI を実行できるようにするには、[System Preferences] (システム環境設定) > [Security & Privacy] (セキュリティとプライバシー) に移動します。テストを実行するときは、これをさらに 3 回実行することが必要になる場合があります。

   IDT-FreeRTOS UI がデフォルトのブラウザで開きます。サポートされるブラウザの詳細については、サポートされているウェブブラウザを使用するを参照してください。

新しい設定を作成する

初めて使用する場合、IDT-FreeRTOS がテストを実行するのに必要な JSON 設定ファイルをセットアップするための新しい設定を作成する必要があります。これにより、テストを実行したり、作成した設定を変更できます。

`config.json`、`device.json`、および `userdata.json` ファイルの例については、マイクロコントローラーボードの最初のテストを参照してください。Bluetooth Low Energy (BLE) テストにのみ使用する `resource.json` ファイルの例については、Bluetooth Low Energy テストを実行するを参照してください。

新しい設定を作成するには

1. IDT-FreeRTOS UI でナビゲーションメニューを開いて、[Create new configuration] (新しい設定の作成) を選択します。

   > **Important**
   >
   > UI を開く前に、 AWS 認証情報を設定する必要があります。認証情報を設定していない場合は、IDT-FreeRTOS UI ブラウザウィンドウを閉じて、AWS 認証情報を設定するの手順に従います。その後に IDT-FreeRTOS UI を再度開きます。

2. 設定ウィザードに従って、認定テストの実行に使用される IDT 構成設定を入力します。このウィザードで *`devicetester-extract-location`*`/config` ディレクトリにある JSON 設定ファイル内の次の設定を行います。

   - AWS 設定 — IDT-FreeRTOS AWS アカウント がテスト実行中に AWS リソースを作成するために使用する情報。これらの設定は `config.json` ファイルで行われます。
   - [FreeRTOS repository] (FreeRTOS リポジトリ): FreeRTOS のリポジトリと移植されたコードへの絶対パス、および実行する認定のタイプ。これらの設定は `userdata.json` ファイルで行われます。

     認定テストを実行するには、デバイス用に FreeRTOS を移植する必要があります。詳細については、FreeRTOS 移植ガイドを参照してください。
   - [Build and flash] (ビルドとフラッシュ): IDT がボード上でテストを自動的にビルドおよびフラッシュできるようにする、ハードウェア用のビルドコマンドとフラッシュコマンド。これらの設定は `userdata.json` ファイルで行われます。

- [Devices] (デバイス): テスト対象のデバイス用のデバイスプール設定。これらの設定は、`device.json` ファイルのデバイスプールに関する `id` および `sku` フィールドと `devices` ブロックで行われます。
- [Networking] (ネットワーク): デバイスのネットワーク通信サポートをテストするための設定。これらの設定は、`device.json` ファイルの `features` ブロックと `userdata.json` ファイルの `clientWifiConfig` および `testWifiConfig` ブロックで行われます。
- [Echo server] (エコーサーバー): セキュアソケットテスト用のエコーサーバー設定。これらの設定は `userdata.json` ファイルで行われます。

  エコーサーバー設定の詳細については、https://docs.aws.amazon.com/freertos/latest/portingguide/afr-echo-server.htmlを参照してください。
- [CMake]: (オプション) CMake ビルド機能テストを実行するための設定。この設定は、CMake をビルドシステムとして使用する場合にのみ必要です。これらの設定は `userdata.json` ファイルで行われます。
- [BLE]: Bluetooth Low Energy 機能テストを実行するための設定。これらの設定は、`device.json` ファイルの `features` ブロックおよび `resource.json` ファイルで行われます。
- [OTA]: OTA 機能テストを実行するための設定。これらの設定は、`device.json` ファイルの `features` ブロックおよび `userdata.json` ファイルで行われます。

3. [Review] (確認) ページで、設定情報を確認します。

設定の確認が完了したら、[Run tests] (テストの実行) を選択して、認定テストを実行します。

既存の設定を変更する

IDT の設定ファイルを既にセットアップしている場合は、IDT-FreeRTOS UI を使用して既存の設定を変更できます。既存の設定ファイルが *`devicetester-extract-location`*`/config` ディレクトリにあることを確認します。

新しい設定を変更するには

1. IDT-FreeRTOS UI でナビゲーションメニューを開いて、[Edit existing configuration] (既存の設定の編集) を選択します。

   設定ダッシュボードには、既存の構成設定に関する情報が表示されます。設定が正しくない、または使用できない場合、設定のステータスは `Error validating configuration` です。
2. 既存の設定を変更するには、以下のステップを実行します。

a. 構成設定の名前を選択して、設定ページを開きます。

b. 設定を変更し、[Save] (保存) を選択して、対応する設定ファイルを再生成します。

設定の変更が完了したら、すべての構成設定が検証に合格することを確認します。各構成設定のステータスが `Valid` の場合、この設定を使用して認定テストを実行できます。

認定テストを実行する

IDT-FreeRTOS の設定を作成したら、認定テストを実行できます。

認定テストを実行するには

1. 設定を確認します。
2. ナビゲーションメニューで、[Run tests] (テストの実行) を選択します。
3. テストの実行を開始するには、[Start tests] (テストの開始) を選択します。

IDT-FreeRTOS は認定テストを実行し、テスト実行の概要とエラーを Test runner コンソールに表示します。テストの実行が完了したら、次の場所でテスト結果とログを確認できます。

- テスト結果は、*devicetester-extract-location*`/results/`*execution-id* ディレクトリにあります。
- テストのログは *devicetester-extract-location*`/results/`*execution-id*`/logs` ディレクトリにあります。

テスト結果とログの詳細については、「」およびIDT for FreeRTOS の結果を表示する「」を参照してくださいIDT for FreeRTOS ログを表示する。

Bluetooth Low Energy テストを実行する

このセクションでは、 AWS IoT Device Tester for FreeRTOS を使用して Bluetooth Low Energy テストをセットアップして実行する方法について説明します。

コア資格には Bluetooth テストの必要はありません。FreeRTOS Bluetooth サポートを使用してデバイスをテストしない場合はこのセットアップをスキップできますが、device.json の BLE 機能を No に設定していることを確認してください。

前提条件

- 「マイクロコントローラーボードの最初のテスト」の手順に従います。
- Raspberry Pi 4B または 3B+。(Raspberry Pi BLE コンパニオンアプリケーションを実行するために必要)
- Raspberry Pi のソフトウェア用のマイクロ SD カードおよび SD カードアダプタ。

Raspberry Pi のセットアップ

テスト対象デバイス (DUT) の BLE 機能をテストするには、Raspberry Pi Model 4B または 3B+ を保有している必要があります。

BLE テストを実行するように Raspberry Pi をセットアップするには

1. テストの実行に必要なソフトウェアが含まれているカスタム Yocto イメージのいずれかをダウンロードします。

   - Raspberry Pi 4B 用のイメージ
   - Raspberry Pi 3B+ 用のイメージ

   > **Note**
   >
   > Yocto イメージは AWS IoT Device Tester 、 for FreeRTOS でのテストにのみ使用し、他の目的では使用しないでください。

2. Raspberry Pi 用の SD カードに yocto イメージをフラッシュします。

   - Etcher などの SD カード書き込みツールを使用して、ダウンロードした *image-name*`.rpi-sd.img` ファイルを SD カードにフラッシュします。オペレーティングシステムイメージが大きいため、このステップに時間がかかることがあります。次に、SD カードをコンピュータから取り出し、microSD カードを Raspberry Pi に挿入します。

3. Raspberry Pi を設定します。

   a. 最初の起動では、Raspberry Pi をモニター、キーボード、およびマウスに接続することをお勧めします。

   b. Raspberry Pi をマイクロ USB 電源に接続します。

c. デフォルトの認証情報を使用してサインインします。ユーザー ID には **root** と入力します。パスワードには **idtafr** と入力します。

d. イーサネットまたは Wi-Fi 接続を使用して、Raspberry Pi をネットワークに接続します。

   i. Wi-Fi 経由で Raspberry Pi を接続するには、Raspberry Pi 上で `/etc/wpa_supplicant.conf` を開き、`Network` 接続に Wi-Fi 接続を追加します。

```
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. `ifup wlan0` を実行して Wi-Fi 接続を開始します。Wi-Fi ネットワークに接続するには 1 分かかる場合があります。

e. イーサネット接続の場合は、`ifconfig eth0` を実行します。Wi-Fi 接続の場合は、`ifconfig wlan0` を実行します。コマンドの出力に `inet addr` として表示される IP アドレスを書き留めます。この手順の後半でこの IP アドレスが必要になります。

f. (オプション) このテストは、yocto イメージ用のデフォルトの認証情報を使用して SSH を介して Raspberry Pi 上でコマンドを実行します。より高度なセキュリティには、SSH にパブリックキー認証を設定し、パスワードベースの SSH を無効にすることが推奨されます。

   i. OpenSSL `ssh-keygen` コマンドを使用して SSH キーを作成します。ホストコンピュータに SSK キーペアがすでにある場合は、 AWS IoT Device Tester FreeRTOS が Raspberry Pi にサインインできるように新しいキーペアを作成することをお勧めします。

> **Note**
>
> Windows にはインストール済みの SSH クライアントがありません。Windows に SSH クライアントをインストールする方法については、「Windows に SSH ソフトウェアをダウンロードする」を参照してください。

   ii. `ssh-keygen` コマンドは、キーペアを保存する名前とパスの入力を求めます。デフォルトでは、キーペアファイルの名前は `id_rsa` (プライベートキー) と `id_rsa.pub`

(パブリックキー) です。macOS および Linux の場合、これらのファイルのデフォルトの場所は ~/.ssh/ です。Windows の場合、デフォルトの場所は C:\Users*user-name* です。

iii. キーフレーズの入力を求められたら、単にエンターキーを押します。

iv. AWS IoT Device Tester for FreeRTOS がデバイスにサインインできるように Raspberry Pi に SSH キーを追加するには、ホストコンピュータから ssh-copy-id コマンドを使用します。このコマンドは、Raspberry Pi の ~/.ssh/authorized_keys ファイルにパブリックキーを追加します。

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

v. パスワードの入力を求められたら、「**idtafr**」と入力します。これは yocto イメージのデフォルトのパスワードです。

> **Note**
>
> ssh-copy-id コマンドが引き受けるパブリックキーの名前は id_rsa.pub です。macOS および Linux では、デフォルトの場所は ~/.ssh/ になります。Windows の場合、デフォルトの場所は C:\Users*user-name*\.ssh です。パブリックキーに別の名前や別の保存先を指定した場合は、ssh-copy-id で -i オプションを使用し、SSH 公開鍵への完全修飾パス (ssh-copy-id -i ~/my/path/myKey.pub など) を指定する必要があります。SSH キーの作成とパブリックキーのコピーの詳細については、「SSH-COPY-ID」を参照してください。

vi. パブリックキー認証が動作していることをテストするには、ssh -i */my/path/myKey* root@*raspberry-pi-device-ip* を実行します。

パスワードの入力を求められない場合、パブリックキー認証が動作しています。

vii. パブリックキーを使用して Raspberry Pi にサインインできることを確認したら、パスワードベースの SSH を無効にします。

A. Raspberry Pi で /etc/ssh/sshd_config ファイルを編集します。

B. PasswordAuthentication 属性を no に設定します。

C. sshd_config ファイルを保存して閉じます。

D. /etc/init.d/sshd reload を実行して SSH サーバーを再ロードします。

g. resource.json ファイルを作成します。

i. AWS IoT Device Tester を抽出したディレクトリで、 という名前のファイルを作成しますresource.json。

ii. Raspberry Pi に関する次の情報をこのファイルに追加し、*rasp-pi-ip-address* を Raspberry Pi の IP アドレスで置き換えます。

```
[
    {
        "id": "ble-test-raspberry-pi",
        "features": [
            {"name":"ble", "version":"4.2"}
        ],
        "devices": [
            {
                "id": "ble-test-raspberry-pi-1",
                "connectivity": {
                    "protocol": "ssh",
                    "ip": "rasp-pi-ip-address"
                }
            }
        ]
    }
]
```

iii. SSH にパブリックキー認証の使用を選択しなかった場合、resource.json ファイルの connectivity セクションに以下を追加します。

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

iv. (オプション) SSH にパブリックキーの使用を選択した場合、resource.json ファイルの connectivity セクションに以下を追加します。

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

FreeRTOS デバイスのセットアップ

device.json ファイルで、BLE 機能を Yes に設定します。Bluetooth テストが利用可能になる前に device.json ファイルを開始した場合、BLE 用の機能を features 配列に追加する必要があります。

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

BLE テストを実行する

device.json で BLE 機能を有効にすると、グループ ID を指定せずに devicetester_*[linux | mac | win_x86-64]* run-suite を実行したときに BLE テストが実行されます。

BLE テストを個別に実行する場合、次のように BLE の グループ ID を指定できます。devicetester_*[linux | mac | win_x86-64]* run-suite --userdata *path-to-userdata*/userdata.json --group-id FullBLE。

より高い信頼性のあるパフォーマンスには、Raspberry Pi をテスト対象のデバイス (DUT) に近い位置に配置します。

BLE テストのトラブルシューティング

「マイクロコントローラーボードの最初のテスト」のステップを実行したことを確認します。BLE 以外のテストが失敗した場合、Bluetooth 設定がこの問題の原因である可能性はほとんどありません。

FreeRTOS 認定スイートを実行する

AWS IoT Device Tester for FreeRTOS 実行可能ファイルを使用して IDT for FreeRTOS とやり取りします。以下のコマンドラインの例では、デバイスプール (同一デバイスの集合) に対して適格性確認テストを実行する方法を示します。

IDT v3.0.0 and later

```
devicetester_[linux | mac | win] run-suite  \
    --suite-id suite-id  \
    --group-id group-id  \
    --pool-id your-device-pool \
    --test-id test-id  \
    --upgrade-test-suite y|n  \
    --update-idt y|n  \
    --update-managed-policy y|n  \
    --userdata userdata.json
```

デバイスプールに対してテストスイートを実行します。userdata.json ファイルは、*devicetester_extract_location*/devicetester_afreertos_*[win|mac|linux]*/configs/ ディレクトリに置く必要があります。

> **Note**
>
> Windows 上で IDT for FreeRTOS を実行する場合、スラッシュ (/) を使用して userdata.json ファイルへのパスを指定します。

特定のテストグループを実行するには、次のコマンドを使用します。

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

```
    --userdata userdata.json
```

1 つのデバイスプールに対して 1 つのスイートを実行する場合 (device.json ファイルで定義したデバイスプールが 1 つしかない場合)、suite-id パラメータと pool-id パラメータは省略可能です。

テストグループの特定のテストケースを実行するには、次のコマンドを使用します。

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

list-test-cases コマンドを使用して、テストグループのテストケースを一覧表示できます。

IDT for FreeRTOS のコマンドラインオプション

group-id

(オプション) 実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。

pool-id

(オプション) テストするデバイスプール。これは、device.json で複数のデバイスプールを定義する場合に必要です。デバイスプールが 1 つしかない場合は、このオプションを省略できます。

suite-id

(オプション) 実行するテストスイートのバージョン。指定しない場合、IDT はシステムの tests ディレクトリにある最新バージョンを使用します。

> **Note**
>
> IDT v3.0.0 以降、IDT は新しいテストスイートをオンラインでチェックします。詳細については、テストスイートのバージョン を参照してください。

test-id

(オプション) 実行するテスト (カンマ区切りリストとして)。指定した場合、group-id は 1 つのグループを指定する必要があります。

Example

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

update-idt

(オプション) このパラメータが設定されておらず、新しい IDT バージョンが使用可能な場合は、IDT の更新を求められます。このパラメータが Y に設定されている場合、IDT は利用可能な新しいバージョンを検出すると、テストの実行を停止します。このパラメータが N に設定されている場合、IDT はテストの実行を継続します。

update-managed-policy

(オプション) このパラメータを使用せず、IDT がマネージドポリシーが最新でないことを検出すると、マネージドポリシーの更新を求められます。このパラメータが Y に設定されている場合、IDT はマネージドポリシーが最新でないことを検出すると、テストの実行を停止します。このパラメータが N に設定されている場合、IDT はテストの実行を継続します。

upgrade-test-suite

(オプション) これを使用なくても、新しいテストスイートのバージョンが使用可能な場合は、ダウンロードするよう求められます。プロンプトを非表示にするには、n を指定して常に最新のテストスイートをダウンロードするか、y を指定して指定したテストスイートまたはシステム上の最新バージョンを使用します 。

Example

例

常に最新のテストスイートをダウンロードして使用するには、次のコマンドを使用します。

```
devicetester_[linux | mac | win_x86-64] run-suite --userdata userdata file --
group-id group ID --upgrade-test-suite y
```

システムで最新のテストスイートを使用するには、次のコマンドを使用します。

```
devicetester_[linux | mac | win_x86-64] run-suite --userdata userdata file --
group-id group ID --upgrade-test-suite n
```

h

`run-suite` オプションの詳細を確認するには、ヘルプオプションを使用します。

Example

例

```
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
```

userdata.json ファイルは、*devicetester_extract_location*/devicetester_afreertos_*[win|mac|linux]*/configs/ ディレクトリになければなりません。

> **Note**
>
> Windows 上で IDT for FreeRTOS を実行する場合、スラッシュ (/) を使用して userdata.json ファイルへのパスを指定します。

特定のテストグループを実行するには、次のコマンドを使用します。

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

1 つのデバイスプールに対して 1 つのスイートを実行する場合 (device.json ファイルで定義したデバイスプールが 1 つしかない場合)、suite-id と pool-id は省略可能です。

IDT for FreeRTOS のコマンドラインオプション

group-id

(オプション) テストグループを指定します。

pool-id

テストするデバイスプールを指定します。デバイスプールが 1 つしかない場合は、このオプションを省略できます。

suite-id

(オプション) 実行するテストスイートを指定します。

IDT for FreeRTOS コマンド

IDT for FreeRTOS のコマンドは次のオペレーションをサポートしています。

IDT v3.0.0 and later

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のスイート内のグループを一覧表示します。

`list-suites`

使用可能なスイートを一覧表示します。

`list-supported-products`

サポートされている製品とテストスイートのバージョンを一覧表示します。

`list-supported-versions`

現在の IDT バージョンでサポートされている FreeRTOS およびテストスイートのバージョンを一覧表示します。

`list-test-cases`

指定したグループのテストケースを一覧表示します。

`run-suite`

デバイスプールに対してテストスイートを実行します。

`--suite-id` オプションを使用してテストスイートのバージョンを指定するか、そのオプションを省略してシステムの最新バージョンを使用します。

個々のテストケースを実行するには、`--test-id` を使用します。

Example

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

オプションの完全な一覧については、「FreeRTOS 認定スイートを実行する」を参照してください。

> **Note**
>
> IDT v3.0.0 以降、IDT は新しいテストスイートをオンラインでチェックします。詳細については、テストスイートのバージョン を参照してください。

IDT v1.7.0 and earlier

`help`

指定されたコマンドに関する情報を一覧表示します。

`list-groups`

特定のスイート内のグループを一覧表示します。

`list-suites`

使用可能なスイートを一覧表示します。

`run-suite`

デバイスプールに対してテストスイートを実行します。

再資格のテスト

IDT for FreeRTOS 認定テストの新しいバージョンがリリースされた場合や、ボード固有のパッケージまたはデバイスドライバーを更新した場合は、IDT for FreeRTOS を使用してマイクロコントローラーボードをテストできます。その後の認定テストでは、最新バージョンの FreeRTOS と IDT for FreeRTOS を使用してテストを再実行してください。

IDT for FreeRTOS の結果を表示する

実行中、IDT はコンソール、ログファイル、テストレポートにエラーを書き込みます。IDT で適格性テストスイートを実行すると、テスト実行の概要がコンソールに書き込まれ、2 つのレポートが生成

されます。これらのレポートは *devicetester-extract-location*/results/*execution-id*/ にあります。両レポートでは、適格性確認テストスイートの実行の結果をキャプチャします。

awsiotdevicetester_report.xml は、 AWS Partner Device Catalog にデバイスを一覧表示 AWS するために に送信する認定テストレポートです。レポートには、次の要素が含まれます。

- IDT for FreeRTOS のバージョン。
- テスト済みの FreeRTOS のバージョン。
- 合格したテストに基づいてデバイスでサポートされる FreeRTOS の機能。
- device.json ファイルに記載されている SKU とデバイス名。
- device.json ファイルに記載されているデバイスの機能。
- テストケース結果の要約を集計したもの。
- デバイスの機能 (FullWiFi、FullMQTT など) に基づいてテストしたライブラリごとのテストケース結果の内訳。
- この FreeRTOS の認定が LTS ライブラリを使用するバージョン 202012.00 を対象としているかどうか。

FRQ_Report.xml は、標準 JUnit XML 形式のレポートです。Jenkins、Bamboo など CI/CD プラットフォームに統合することができます。レポートには、次の要素が含まれます。

- テストケース結果の要約を集計したもの。
- デバイスの機能に基づいてテストしたライブラリごとのテストケース結果の内訳。

IDT for FreeRTOS の結果の解釈

awsiotdevicetester_report.xml または FRQ_Report.xml のレポートセクションには、実行したテストの結果が一覧表示されます。

最初の XML タグ <testsuites> は、テストの実行の概要を含みます。例:

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

<testsuites> タグで使用される属性

name

テストスイートの名前。

time

適格性確認スイートの実行所要時間 (秒)。

tests

実行されたテストケースの数。

failures

実行されたテストケースのうち、合格しなかったものの数。

errors

IDT for FreeRTOS が実行できなかったテストケースの数。

disabled

この属性は使用されていないため無視できます。

テストケースの障害やエラーがない場合、デバイスは FreeRTOS を実行するための技術要件を満たし、 AWS IoT サービスと相互運用できます。 AWS Partner Device Catalog にデバイスを一覧表示することを選択した場合は、このレポートを認定の証拠として使用できます。

テストケースに障害やエラーが発生した場合は、<testsuites> XML タグを確認することで、障害の生じたテストケースを特定できます。<testsuites> タグ内の <testsuite> XML タグは、テストグループのテストケース結果の要約を示します。

```
<testsuite name="FullMQTT" package="" tests="16" failures="0" time="76"
disabled="0" errors="0" skipped="0">
```

形式は <testsuites> タグと似ていますが、skipped という属性があります。この属性は使用されていないため無視できます。<testsuite> XML タグの内側には、テストグループに対して実行されたそれぞれのテストケースの <testcase> タグがあります。例:

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase"
attempts="1"></testcase>
```

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

version

テスト対象の製品のバージョン。

sdk

SDK で IDT を実行した場合、このブロックには SDK の名前とバージョンが含まれます。SDK で IDT を実行しなかった場合、このブロックには以下が含まれます。

```
<sdk>
    <name>N/A</vame>
    <version>N/A</version>
</sdk>
```

features

検証された機能です。`required` としてマークされている機能については、資格を得るためにボードを提出するために必要です。次のスニペットは、この情報が `awsiotdevicetester_report.xml` ファイルで表示される方法を示します。

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

`optional` としてマークされている機能は、資格を得るために必要ありません。次のスニペットは、オプションの機能を示しています。

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

必要な機能のテストに障害やエラーがない場合、そのデバイスは FreeRTOS を実行するための技術的要件を満たしており、 AWS IoT サービスとの相互運用が可能です。AWS Partner Device Catalog にデバイスを出品する場合は、認定の証拠としてこのレポートを使用できます。

テストに障害やエラーが発生した場合は、`<testsuites>` XML タグを確認することで、障害の生じたテストを特定できます。`<testsuites>` タグ内の `<testsuite>` XML タグは、テストグループのテスト結果の要約を示します。例:

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

形式は <testsuites> タグと似ていますが、使用されていないため無視できる skipped という属性があります。各 <testsuite> XML タグ内には、テストグループの実行されたテスト別の <testcase> タグがあります。例:

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

lts

LTS ライブラリを使用する FreeRTOS のバージョンを対象とする場合は true、それ以外の場合は false。

<testcase> タグで使用される属性

name

テストケースの名前。

attempts

IDT for FreeRTOS がテストケースを実行した回数。

テストに障害やエラーが発生した場合、<failure> タグまたは <error> タグがトラブルシューティングのための情報とともに <testcase> タグに追加されます。例:

```
<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>
```

詳細については、「エラーのトラブルシューティング」を参照してください。

IDT for FreeRTOS ログを表示する

テストの実行中に IDT for FreeRTOS によって生成されるログは、*devicetester-extract-location*/results/*execution-id*/logs にあります。2 組のログが生成されます。

test_manager.log

IDT for FreeRTOS から生成されるログ (設定とレポート生成に関連するログなど) を含みます。

***test_group_id*__*test_case_id*.log (FullMQTT__Full_MQTT.log など)**

テスト対象のデバイスからの出力を含む、テストケースのログファイル。ログファイルには、実行されたテストグループとテストケースに従って名前が付けられます。

独自の IDT テストスイートを開発して実行する

IDT v4.0.0 以降、IDT for FreeRTOS では、標準化された構成設定および結果形式と、デバイスやデバイスソフトウェア用のカスタムテストスイートを開発できるテストスイート環境が統合されています。独自の内部検証用のカスタムテストを追加したり、デバイス検証のためにこれらのテストを顧客に提供したりできます。

IDT を使用してカスタムテストスイートを開発および実行するには、次の手順を実行します。

カスタムテストスイートを開発するには

- テストするデバイス用のカスタムテストロジックを使用して、テストスイートを作成します。
- IDT と作成したカスタムテストスイートをテストの実行者に提供します。作成したテストスイートの構成設定に関する情報も提供します。

カスタムテストスイートを実行するには

- テストするデバイスをセットアップします。
- 使用するテストスイートに必要な構成設定を実装します。
- IDT を使用して、カスタムテストスイートを実行します。
- IDT によって実行されたテストのテスト結果と実行ログを表示します。

AWS IoT Device Tester for FreeRTOS の最新バージョンをダウンロードする

IDT の最新バージョンをダウンロードし、ファイルシステム上で読み取りおよび書き込みアクセス許可を持っている場所に抽出します。

> **Note**
>
> 複数のユーザーが NFS ディレクトリや Windows ネットワーク共有フォルダなどの共有場所から IDT を実行することはお勧めしません。IDT パッケージをローカルドライブに展開し、ローカルワークステーションで IDT バイナリを実行することをお勧めします。

Windows では、パスの長さは 260 文字に制限されています。Windows を使用している場合は、パスが 260 文字以内になるようにして、IDT をルートディレクトリ (`C:\` または `D:\` など) に展開します。

テストスイートワークフロー

テストスイートは 3 つのタイプのファイルで設定されます。

- IDT にテストスイートの実行方法に関する情報を提供する設定ファイル。
- IDT がテストケースの実行に使用するテスト実行可能ファイル。
- テストの実行に必要な追加のファイル。

カスタム IDT テストを作成するには、次の基本的な手順を実行します。

1. テストスイート用の設定ファイルを作成します。
2. テストスイート用のテストロジックが含まれるテストケース実行可能ファイルを作成します。
3. テストスイートを実行するためにテストの実行者に必要な設定情報を検証し、文書化します。
4. IDT が予想通りにテストスイートを実行し、テスト結果を生成できることを確認します。

サンプルカスタムスイートを迅速に構築して実行するには、チュートリアル: サンプル IDT テストスイートを構築して実行するの手順に従ってください。

Python でカスタムテストスイートの作成を開始するには、チュートリアル: シンプルな IDT テストスイートの開発を参照してください。

チュートリアル: サンプル IDT テストスイートを構築して実行する

AWS IoT Device Tester ダウンロードには、サンプルテストスイートのソースコードが含まれています。このチュートリアルを完了すると、サンプルテストスイートを構築して実行し、 AWS IoT Device Tester for FreeRTOS を使用してカスタムテストスイートを実行する方法を理解できます。このチュートリアルでは SSH を使用しますが、 AWS IoT Device Tester FreeRTOS デバイスで を使用する方法を学ぶと便利です。

このチュートリアルでは、次の手順を実行します。

1. サンプルテストスイートを構築する

2. IDT を使用してサンプルテストスイートを実行する

トピック

- サンプルテストスイートの前提条件を設定する
- IDT 用のデバイス情報を設定する
- サンプルテストスイートを構築する
- IDT を使用してサンプルテストスイートを実行する
- エラーのトラブルシューティング

サンプルテストスイートの前提条件を設定する

このチュートリアルを完了するには、以下が必要です。

- ホストコンピュータの要件
  - の最新バージョン AWS IoT Device Tester
  - Python 3.7 以降

    コンピュータにインストールされている Python のバージョンを確認するには、次のコマンドを実行します。

    ```
    python3 --version
    ```

    Windows で、このコマンドを使用してエラーが返された場合は、代わりに `python --version` を使用してください。返されたバージョン番号が 3.7 以上の場合は、Powershell ターミナルで次のコマンドを実行し、`python3` を `python` コマンドのエイリアスとして設定します。

    ```
    Set-Alias -Name "python3" -Value "python"
    ```

    バージョン情報が返されない場合や、バージョン番号が 3.7 未満 の場合は、Python のダウンロードの手順に従って Python 3.7 以上をインストールしてください。詳細については、Python のドキュメントを参照してください。
  - urllib3

    `urllib3` が正しくインストールされていることを確認するには、次のコマンドを実行します。

```
python3 -c 'import urllib3'
```

urllib3 がインストールされていない場合は、次のコマンドを実行してインストールします。

```
python3 -m pip install urllib3
```

- デバイスの要件
  - Linux オペレーティングシステムが搭載され、ホストコンピュータと同じネットワークにネットワーク接続するデバイス。

    Raspberry Pi OS が搭載された Raspberry Pi を使用することをお勧めします。Raspberry Pi に SSH をセットアップし、リモートから接続できることを確認します。

IDT 用のデバイス情報を設定する

IDT がテストを実行するためのデバイス情報を設定します。次の情報を使用して、*<device-tester-extract-location>*/configs フォルダに含まれている device.json テンプレートを更新する必要があります。

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
```

```
    ]
  }
]
```

devices オブジェクトで、次の情報を指定します。

id

自分のデバイスのユーザー定義の一意の識別子。

connectivity.ip

自分のデバイスの IP アドレス。

connectivity.port

オプション。デバイスへの SSH 接続に使用するポート番号。

connectivity.auth

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.user

デバイスへのサインインに使用するユーザー名。

connectivity.auth.credentials.privKeyPath

デバイスへのサインインに使用するプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

devices.connectivity.auth.credentials.password

自分のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

> **Note**
>
> method が pki に設定されている場合のみ privKeyPath を指定します。
> method が password に設定されている場合のみ password を指定します。

サンプルテストスイートを構築する

<device-tester-extract-location>/samples/python フォルダには、サンプル設定ファイル、ソースコード、および提供されたビルドスクリプトを使用してテストスイートに結合できる IDT クライアント SDK が含まれています。次のディレクトリツリーは、これらのサンプルファイルの場所を示しています。

```
<device-tester-extract-location>
### ...
### tests
### samples
#   ### ...
#   ### python
#       ### configuration
#       ### src
#       ### build-scripts
#           ### build.sh
#           ### build.ps1
### sdks
    ### ...
    ### python
        ### idt_client
```

テストスイートを構築するには、ホストコンピュータで次のコマンドを実行します。

Windows

```
cd <device-tester-extract-location>/samples/python/build-scripts
```

```
./build.ps1
```

Linux, macOS, or UNIX

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.sh
```

これにより、*<device-tester-extract-location>*/tests フォルダ内の IDTSampleSuitePython_1.0.0 フォルダにサンプルテストスイートが作成されます。IDTSampleSuitePython_1.0.0 フォルダ内のファイルを確認して、サンプルテストスイートの構造を理解し、テストケース実行可能ファイルとテスト設定ファイルのさまざまな例を参照してください。

> **Note**
>
> サンプルテストスイートには python ソースコードが含まれます。テストスイートのコードには機密情報を入力しないでください。

次のステップ: IDT を使用して、作成したサンプルテストスイートを実行します。

IDT を使用してサンプルテストスイートを実行する

サンプルテストスイートを実行するには、ホストコンピュータで次のコマンドを実行します。

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id IDTSampleSuitePython
```

IDT はサンプルテストスイートを実行し、結果をコンソールにストリーミングします。テストの実行が完了すると、次の情報が表示されます。

```
========== Test Summary ==========
Execution Time:         5s
Tests Completed:        4
Tests Passed:           4
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    sample_group:       PASSED
```

```
---------------------------------
Path to AWS IoT Device Tester Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/IDTSampleSuitePython_Report.xml
```

エラーのトラブルシューティング

次の情報は、チュートリアルの実行に関連する問題の解決に役立ちます。

テストケースが正常に実行されない

- テストが正常に実行されない場合、IDT はエラーログをコンソールにストリーミングします。このログはテスト実行のトラブルシューティングに役立ちます。このチュートリアルのすべての前提条件を満たしていることを確認してください。

テスト対象のデバイスに接続できない

以下について確認します。

- `device.json` ファイルに、正しい IP アドレス、ポート、および認証情報が含まれている。
- ホストコンピュータから SSH 経由でデバイスに接続できる。

チュートリアル: シンプルな IDT テストスイートの開発

テストスイートは、以下を組み合わせたものです。

- テストロジックを含むテスト実行可能ファイル
- テストスイートについて記述する設定ファイル

このチュートリアルでは、IDT for FreeRTOS を使用して、単一のテストケースを含む Python テストスイートを開発する方法を説明します。このチュートリアルでは SSH を使用しますが、 AWS IoT Device Tester FreeRTOS デバイスで を使用する方法を学ぶと便利です。

このチュートリアルでは、次の手順を実行します。

1. テストスイートディレクトリを作成する

2. 設定ファイルを作成する
3. テストケース実行可能ファイルを作成する
4. テストスイートを実行する

以下のステップに従って、シンプルな IDT テストスイートを開発するためのチュートリアルを完了します。

トピック

- シンプルな IDT テストスイートの前提条件を設定する
- テストスイートディレクトリを作成する
- 設定ファイルを作成する
- IDT クライアント SDK を入手する
- テストケース実行可能ファイルを作成する
- IDT 用のデバイス情報を設定する
- テストスイートを実行する
- エラーのトラブルシューティング
- IDT テストスイート設定ファイルを作成する
- IDT テストオーケストレーターを設定する
- IDT ステートマシンを設定する
- IDT テストケース実行可能ファイルを作成する
- IDT コンテキストを使用する
- テストの実行者向けの設定の構成
- カスタムテストスイートのデバッグと実行
- IDT テストの結果とログを確認する
- IDT 使用状況メトリクスを送信する

シンプルな IDT テストスイートの前提条件を設定する

このチュートリアルを完了するには、以下が必要です。

- ホストコンピュータの要件
  - の最新バージョン AWS IoT Device Tester
  - Python 3.7 以降

コンピュータにインストールされている Python のバージョンを確認するには、次のコマンドを実行します。

```
python3 --version
```

Windows で、このコマンドを使用してエラーが返された場合は、代わりに `python --version` を使用してください。返されたバージョン番号が 3.7 以上の場合は、Powershell ターミナルで次のコマンドを実行し、`python3` を `python` コマンドのエイリアスとして設定します。

```
Set-Alias -Name "python3" -Value "python"
```

バージョン情報が返されない場合や、バージョン番号が 3.7 未満 の場合は、Python のダウンロードの手順に従って Python 3.7 以上をインストールしてください。詳細については、Python のドキュメントを参照してください。

- urllib3

  `urllib3` が正しくインストールされていることを確認するには、次のコマンドを実行します。

  ```
  python3 -c 'import urllib3'
  ```

  `urllib3` がインストールされていない場合は、次のコマンドを実行してインストールします。

  ```
  python3 -m pip install urllib3
  ```

- デバイスの要件
  - Linux オペレーティングシステムが搭載され、ホストコンピュータと同じネットワークにネットワーク接続するデバイス。

    Raspberry Pi OS が搭載された Raspberry Pi を使用することをお勧めします。Raspberry Pi に SSH をセットアップし、リモートから接続できることを確認します。

テストスイートディレクトリを作成する

IDT は、テストケースを、各テストスイート内のテストグループに論理的に分離します。
各テストケースはテストグループ内に存在する必要があります。このチュートリアルで

は、MyTestSuite_1.0.0 という名前のフォルダを作成し、このフォルダ内に次のディレクトリツリーを作成します。

```
MyTestSuite_1.0.0
### suite
    ### myTestGroup
        ### myTestCase
```

設定ファイルを作成する

テストスイートには、次の必須の設定ファイルが含まれている必要があります。

必要なファイル

suite.json

テストスイートに関する情報が含まれています。「suite.json を設定する」を参照してください。

group.json

テストグループに関する情報が含まれています。テストスイート内のテストグループごとに group.json ファイルを作成する必要があります。「group.json を設定する」を参照してください。

test.json

テストケースに関する情報が含まれています。テストスイート内のテストケースごとに test.json ファイルを作成する必要があります。「test.json を設定する」を参照してください。

1. MyTestSuite_1.0.0/suite フォルダで、次の構造の suite.json ファイルを作成します。

```
{
    "id": "MyTestSuite_1.0.0",
    "title": "My Test Suite",
    "details": "This is my test suite.",
    "userDataRequired": false
}
```

2. MyTestSuite_1.0.0/myTestGroup フォルダで、次の構造の group.json ファイルを作成します。

```
{
    "id": "MyTestGroup",
    "title": "My Test Group",
    "details": "This is my test group.",
    "optional": false
}
```

3. MyTestSuite_1.0.0/myTestGroup/myTestCase フォルダで、次の構造の test.json ファイルを作成します。

```
{
    "id": "MyTestCase",
    "title": "My Test Case",
    "details": "This is my test case.",
    "execution": {
        "timeout": 300000,
        "linux": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "mac": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "win": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        }
    }
}
```

MyTestSuite_1.0.0 フォルダのディレクトリツリーは次のようになります。

```
MyTestSuite_1.0.0
### suite
```

```
    ### suite.json
    ### myTestGroup
        ### group.json
        ### myTestCase
            ### test.json
```

IDT クライアント SDK を入手する

IDT がテスト対象のデバイスとやり取りし、テスト結果をレポートできるようにするには、IDT クライアント SDK を使用します。このチュートリアルでは、Python バージョンの SDK を使用します。

<device-tester-extract-location>/sdks/python/ フォルダから、idt_client フォルダを自分の MyTestSuite_1.0.0/suite/myTestGroup/myTestCase フォルダにコピーします。

SDK が正常にコピーされたことを確認するには、次のコマンドを実行します。

```
cd MyTestSuite_1.0.0/suite/myTestGroup/myTestCase
python3 -c 'import idt_client'
```

テストケース実行可能ファイルを作成する

テストケース実行可能ファイルには、実行するテストロジックが含まれています。テストスイートには、複数のテストケース実行可能ファイルを含めることができます。このチュートリアルでは、テストケース実行可能ファイルを 1 つだけ作成します。

1. テストスイートファイルを作成します。

   MyTestSuite_1.0.0/suite/myTestGroup/myTestCase フォルダで、次の内容の myTestCase.py ファイルを作成します。

   ```
   from idt_client import *

   def main():
       # Use the client SDK to communicate with IDT
       client = Client()

   if __name__ == "__main__":
       main()
   ```

2. クライアント SDK 関数を使用して、自分の myTestCase.py ファイルに次のテストロジックを追加します。

a. テスト対象のデバイスで SSH コマンドを実行します。

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

if __name__ == "__main__":
    main()
```

b. テスト結果を IDT に送信します。

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

    # Create a send result request
    sr_req = SendResultRequest(TestResult(passed=True))
```

```
    # Send the result
    client.send_result(sr_req)

if __name__ == "__main__":
    main()
```

IDT 用のデバイス情報を設定する

IDT がテストを実行するためのデバイス情報を設定します。次の情報を使用して、*<device-tester-extract-location>*/configs フォルダに含まれている device.json テンプレートを更新する必要があります。

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

devices オブジェクトで、次の情報を指定します。

id

自分のデバイスのユーザー定義の一意の識別子。

connectivity.ip

自分のデバイスの IP アドレス。

connectivity.port

オプション。デバイスへの SSH 接続に使用するポート番号。

connectivity.auth

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.user

デバイスへのサインインに使用するユーザー名。

connectivity.auth.credentials.privKeyPath

デバイスへのサインインに使用するプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

devices.connectivity.auth.credentials.password

自分のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

> **Note**
>
> method が pki に設定されている場合のみ privKeyPath を指定します。
> method が password に設定されている場合のみ password を指定します。

テストスイートを実行する

テストスイートを作成したら、テストスイートが期待どおりに機能することを確認します。そのために、次の手順に従って、既存のデバイスプールを使用してテストスイートを実行します。

1. 自分の MyTestSuite_1.0.0 フォルダを *<device-tester-extract-location>*/tests にコピーします。
2. 以下の コマンドを実行します。

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id MyTestSuite
```

IDT はテストスイートを実行し、結果をコンソールにストリーミングします。テストの実行が完了すると、次の情報が表示されます。

```
time="2020-10-19T09:24:47-07:00" level=info msg=Using pool: pool
time="2020-10-19T09:24:47-07:00" level=info msg=Using test suite "MyTestSuite_1.0.0"
 for execution
time="2020-10-19T09:24:47-07:00" level=info msg=b'hello world\n'
 suiteId=MyTestSuite groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:47-07:00" level=info msg=All tests finished.
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:48-07:00" level=info msg=

========== Test Summary ==========
Execution Time:         1s
Tests Completed:        1
Tests Passed:           1
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    myTestGroup:        PASSED
```

```
---------------------------------
Path to AWS IoT Device Tester Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/MyTestSuite_Report.xml
```

エラーのトラブルシューティング

次の情報は、チュートリアルの実行に関連する問題の解決に役立ちます。

テストケースが正常に実行されない

テストが正常に実行されない場合、IDT はエラーログをコンソールにストリーミングします。このログはテスト実行のトラブルシューティングに役立ちます。エラーログを確認する前に、次の点を確認してください。

- IDT クライアント SDK は、「」で説明されているように、正しいフォルダにありますIDT クライアント SDK を入手する。
- このチュートリアルのすべての前提条件を満たしている。詳細については、「シンプルな IDT テストスイートの前提条件を設定する」を参照してください。

テスト対象のデバイスに接続できない

以下について確認します。

- `device.json` ファイルに、正しい IP アドレス、ポート、および認証情報が含まれている。
- ホストコンピュータから SSH 経由でデバイスに接続できる。

IDT テストスイート設定ファイルを作成する

このセクションでは、カスタムテストスイートの作成時、これに含める設定ファイルを作成する際の形式について説明します。

必要な設定ファイル

`suite.json`

テストスイートに関する情報が含まれています。「suite.json を設定する」を参照してください。

`group.json`

テストグループに関する情報が含まれています。テストスイート内のテストグループごとに `group.json` ファイルを作成する必要があります。「group.json を設定する」を参照してください。

`test.json`

テストケースに関する情報が含まれています。テストスイート内のテストケースごとに `test.json` ファイルを作成する必要があります。「test.json を設定する」を参照してください。

オプションの設定ファイル

`test_orchestrator.yaml`、または `state_machine.json`

IDT がテストスイートを実行するときのテストの実行方法を定義します。「test_orchestrator.yaml を設定する」を参照してください。

> **Note**
>
> IDT v4.5.2 以降では、テストワークフローの定義に `test_orchestrator.yaml` ファイルを使用します。IDT のそれより前のバージョンでは、`state_machine.json`ファイルを開きます。ステートマシンの詳細については、「IDT ステートマシンを設定する」を参照してください。

`userdata_schema.json`

テストの実行者が構成設定に含めることができる `userdata.json` ファイルのスキーマを定義します。`userdata.json` ファイルは、テストの実行に必要であるものの、`device.json` ファイルに含まれていない、追加の設定情報用に使用します。「userdata_schema.json を設定する」を参照してください。

設定ファイルは、以下に示すように *<custom-test-suite-folder>* に配置します。

```
<custom-test-suite-folder>
### suite
    ### suite.json
    ### test_orchestrator.yaml
```

```
    ### userdata_schema.json
    ### <test-group-folder>
        ### group.json
        ### <test-case-folder>
            ### test.json
```

suite.json を設定する

suite.json ファイルは、環境変数を設定し、テストスイートの実行にユーザーデータが必要かどうかを決定します。以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/suite.json ファイルを設定します。

```
{
    "id": "<suite-name>_<suite-version>",
    "title": "<suite-title>",
    "details": "<suite-details>",
    "userDataRequired": true | false,
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        },
        ...
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

id

テストスイートの一意のユーザー定義 ID。id の値は、suite.json ファイルが配置されているテストスイートフォルダ名と一致する必要があります。スイート名とスイートのバージョンは、次の要件も満たしている必要があります。

- *<suite-name>* にアンダースコアを含めることはできません。
- *<suite-version>* が *x.x.x* として表されている (x は数字)。

ID は IDT によって生成されるテストレポートに表示されます。

title

このテストスイートでテストされる製品または機能のユーザー定義名。この名前は、テストの実行者の IDT CLI に表示されます。

details

テストスイートの目的の簡単な説明。

userDataRequired

テストの実行者が userdata.json ファイルにカスタム情報を含める必要があるかどうかを定義します。この値を true に設定した場合は、テストスイートフォルダに userdata_schema.json ファイルも含める必要があります。

environmentVariables

オプション。このテストスイートに設定する環境変数の配列。

environmentVariables.key

環境変数の名前。

environmentVariables.value

環境変数の値。

group.json を設定する

group.json ファイルは、テストグループが必須かオプションかを定義します。以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/*<test-group>*/group.json ファイルを設定します。

```
{
    "id": "<group-id>",
    "title": "<group-title>",
    "details": "<group-details>",
    "optional": true | false,
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

id

テストグループの一意のユーザー定義 ID。id の値は、group.json ファイルが配置されているテストグループフォルダの名前と一致する必要があり、アンダースコア (_) は使用できません。ID は IDT によって生成されるテストレポートで使用されます。

title

テストグループのわかりやすい名前。この名前は、テストの実行者の IDT CLI に表示されます。

details

テストグループの目的の簡単な説明。

optional

オプション。true に設定すると、IDT が必須テストの実行を完了した後に、このテストグループがオプショングループとして表示されます。デフォルト値は false です。

test.json を設定する

test.json ファイルは、テストケースによって使用されるテストケース実行ファイルと環境変数を決定します。テストケース実行可能ファイルの作成の詳細については、IDT テストケース実行可能ファイルを作成するを参照してください。

以下のテンプレートを使用して、*<custom-test-suite-folder>*/suite/*<test-group>*/*<test-case>*/test.json ファイルを設定します。

```
{
    "id": "<test-id>",
    "title": "<test-title>",
    "details": "<test-details>",
    "requireDUT": true | false,
    "requiredResources": [
        {
            "name": "<resource-name>",
            "features": [
                {
                    "name": "<feature-name>",
                    "version": "<feature-version>",
                    "jobSlots": <job-slots>
                }
            ]
        }
```

```
    ],
    "execution": {
        "timeout": <timeout>,
        "mac": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "linux": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "win": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ]
        }
    },
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

id

テストケースの一意のユーザー定義 ID。id の値は、test.json ファイルが配置されているテストケースフォルダの名前と一致する必要があり、アンダースコア (_) は使用できません。ID は IDT によって生成されるテストレポートで使用されます。

title

テストケースのわかりやすい名前。この名前は、テストの実行者の IDT CLI に表示されます。

details

テストケースの目的の簡単な説明。

requireDUT

オプション。このテストの実行にデバイスが必要な場合は `true` に設定します。必要ない場合は `false` に設定します。デフォルト値は `true` です。テストの実行者は、テストの実行に使用するデバイスを `device.json` ファイルに設定します。

requiredResources

オプション。このテストの実行に必要なリソースデバイスに関する情報を指定する配列。

requiredResources.name

このテストの実行時にリソースデバイスに与える一意の名前。

requiredResources.features

ユーザー定義のリソースデバイス機能の配列。

requiredResources.features.name

機能の名前。このデバイスが使用するデバイス機能。この名前は、テストの実行者によって `resource.json` ファイルに指定される機能名に対してマッチングされます。

requiredResources.features.version

オプション。機能のバージョン。この値は、テストの実行者によって `resource.json` ファイルに指定される機能のバージョンに対してマッチングされます。バージョンが指定されていない場合、機能はチェックされません。機能にバージョン番号が必要ない場合は、このフィールドは空白のままにしてください。

requiredResources.features.jobSlots

オプション。この機能がサポートできる同時テストの数。デフォルト値は 1 です。IDT が機能ごとに異なるデバイスを使用する場合は、この値を 1 に設定することをお勧めします。

execution.timeout

IDT がテスト実行終了まで待機する時間 (ミリ秒単位)。この値の設定の詳細については、IDT テストケース実行可能ファイルを作成するを参照してください。

execution.*os*

IDT を実行するホストコンピュータのオペレーティングシステムに基づいて実行されるテストケースの実行可能ファイル。サポートされている値は `linux`、`mac`、`win` です。

execution.*os*.cmd

指定されたオペレーティングシステムで実行するテストケース実行可能ファイルへのパス。この場所は、システムパス内に存在する必要があります。

execution.*os*.args

オプション。テストケースの実行可能ファイルを実行するために指定する引数。

environmentVariables

オプション。このテストケース用に設定された環境変数の配列。

environmentVariables.key

環境変数の名前。

environmentVariables.value

環境変数の値。

> **Note**
>
> test.json ファイルと suite.json ファイルに同じ環境変数を設定した場合は、test.json ファイルの値が優先されます。

test_orchestrator.yaml を設定する

テストオーケストレーターは、テストスイートの実行フローを制御するコンストラクトです。テストスイートの開始ステートを決定し、ユーザー定義のルールに基づいてステートの移行を管理し、終了ステートに達するまでステートの移行を継続します。

テストスイートにユーザー定義のテストオーケストレーターが含まれていない場合は、IDT によってテストオーケストレーターが生成されます。

デフォルトのテストオーケストレーターには以下の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。

- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT テストオーケストレーターの機能の詳細については、「IDT テストオーケストレーターを設定する」を参照してください。

userdata_schema.json を設定する

userdata_schema.json ファイルは、テストの実行者がユーザーデータを指定するスキーマを決定します。ユーザーデータは、テストスイートが device.json ファイルに含まれていない情報を必要とする場合に必要になります。例えば、テストを実行するために、Wi-Fi ネットワークの認証情報、特定のオープンポート、またはユーザーが提供する証明書が必要になる場合があります。この情報は、userdata という入力パラメータとして IDT に提供できます。これは、ユーザーが *<device-tester-extract-location>*/config フォルダに作成するuserdata.json ファイルの値です。userdata.json の形式は、テストケースに含まれている userdata_schema.json ファイルに基づきます。

テストの実行者が userdata.json ファイルを提供しなければならないことを示す方法

1. suite.json ファイルで、userDataRequired を true に設定します。
2. *<custom-test-suite-folder>* で、userdata_schema.json ファイルを作成します。
3. userdata_schema.json ファイルを編集して、有効な IETF Draft v4 JSON Schema を作成します。

IDT は、テストスイートを実行するときに、このスキーマを自動的に読み込み、テストの実行者によって提供される userdata.json ファイルの検証に使用します。有効な場合、userdata.json ファイルのコンテンツは IDT コンテキストおよび、テストオーケストレーターコンテキストの両方で利用可能になります。

IDT テストオーケストレーターを設定する

IDT v4.5.2 以降、IDT には新しいテストオーケストレーターコンポーネントが追加されています。テストオーケストレーターは、テストスイートの実行フローを制御する IDT コンポーネントで、IDT がすべてのテストを実行した後にテストレポートを生成します。テストオーケストレーターは、ユーザー定義のルールに基づいて、テストの選択とテストの実行順序を決定します。

テストスイートにユーザー定義のテストオーケストレーターが含まれていない場合は、IDT によってテストオーケストレーターが生成されます。

デフォルトのテストオーケストレーターには以下の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT ステートマシンは、テストオーケストレーターに置き換えられます。テストスイートの開発には、IDT ステートマシンではなくテストオーケストレーターを使用することを強くお勧めします。テストオーケストレーターでは、以下の機能が改善されています。

- IDT ステートマシンが命令型を使用するのに対し、宣言型を使用します。これにより、どのテストを実行するか、およびいつそれを実行するかを指定できます。
- 特定のグループ処理、レポート生成、エラー処理、結果追跡を管理し、これらのアクションを手動管理する必要がないようにします。
- デフォルトでコメントをサポートする YAML 形式を使用します。
- 同じワークフローを定義するのに必要なディスク容量が、テストオーケストレーターより 80% 少なくなります。
- テスト前検証を追加し、誤ったテスト ID や依存関係の循環がワークフロー定義に含まれていないことを検証します。

テストオーケストレーター形式

次のテンプレートを使用して、独自の *custom-test-suite-folder*/suite/test_orchestrator.yaml ファイルを設定できます。

```
Aliases:
  string: context-expression

ConditionalTests:
  - Condition: context-expression
    Tests:
      - test-descriptor

Order:
```

```
  - - group-descriptor
    - group-descriptor

Features:
  - Name: feature-name
    Value: support-description
    Condition: context-expression
    Tests:
        - test-descriptor
    OneOfTests:
        - test-descriptor
    IsRequired: boolean
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Aliases`

オプション。コンテキスト式にマップするユーザー定義の文字列。エイリアスを使用すると、テストオーケストレーター設定でコンテキスト式を識別するためのフレンドリ名を生成できます。これは、複雑なコンテキスト式や、複数の場所で使用する式を作成する場合に特に便利です。

コンテキスト式を使用するとコンテキストクエリを保存でき、これにより他の IDT 設定からデータにアクセスできるようになります。詳細については、「コンテキスト内のデータにアクセスする」を参照してください。

Example

例

```
Aliases:
    FizzChosen: "'{{$pool.features[?(@.name == 'Fizz')].value[0]}}' == 'yes'"
    BuzzChosen: "'{{$pool.features[?(@.name == 'Buzz')].value[0]}}' == 'yes'"
    FizzBuzzChosen: "'{{$aliases.FizzChosen}}' && '{{$aliases.BuzzChosen}}'"
```

`ConditionalTests`

オプション。条件と、条件が満たされたときに実行される、対応するテストケースのリスト。各条件には複数のテストケースを割り当てることができますが、特定のテストケースを割り当てることができる条件は 1 つだけです。

デフォルトでは、IDT はこのリストの条件に割り当てられていないテストケースをすべて実行します。このセクションを指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。

ConditionalTests リストの各項目には以下のパラメータが含まれます。

Condition

ブール値に評価されるコンテキスト式。評価値が true の場合、IDT は Tests パラメータで指定されたテストケースを実行します。

Tests

テスト記述子のリスト。

各テスト記述子は、テストグループ ID と 1 つ以上のテストケース ID を使用して、特定のテストグループから実行する個々のテストを識別します。テスト記述子は以下の形式を使用します。

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

Example

例

次の例は、Aliases として定義できる汎用コンテキスト式を使用します。

```
ConditionalTests:
    - Condition: "{{$aliases.Condition1}}"
      Tests:
          - GroupId: A
          - GroupId: B
    - Condition: "{{$aliases.Condition2}}"
      Tests:
          - GroupId: D
    - Condition: "{{$aliases.Condition1}} || {{$aliases.Condition2}}"
      Tests:
          - GroupId: C
```

定義された条件に基づき、IDT は次のようにテストグループを選択します。

- Condition1 が真の場合、IDT はテストグループ A、B、C のテストを実行します。
- Condition2 が真の場合、IDT はテストグループ C と D のテストを実行します。

`Order`

オプション。テストを実行する順序。テストの順序はテストのグループレベルで指定します。このセクションを指定しない場合、IDT はすべての適用可能なテストグループをランダムな順序で実行します。`Order` の値は、グループ記述子リストのリストです。`Order` のリストに記載していないテストグループは、記載されている他のテストグループと並行して実行できます。

各グループ記述子リストには 1 つ以上のグループ記述子が含まれ、各記述子で指定されたグループを実行する順序を特定します。個別のグループ記述子を定義するには、以下の形式を使用できます。

- *`group-id`* - 既存のテストグループのグループ ID。
- [*`group-id`*, *`group-id`*] - 相互に任意の順序で実行できるテストグループのリスト。
- "*" - ワイルドカード これは、現在のグループ記述子リストにまだ指定されていないすべてのテストグループのリストに相当します。

`Order` の値は、次の要件も満たしている必要があります。

- グループ記述子で指定するテストグループ ID は、テストスイートに存在する必要があります。
- 各グループ記述子リストには、少なくとも 1 つのテストグループが含まれている必要があります。
- 各グループ記述子リストには一意のグループ ID を含める必要があります。個々のグループ記述子内でテストグループ ID を繰り返すことはできません。
- グループ記述子リストは、最大 1 つのワイルドカードグループ記述子を持つことができます。ワイルドカードグループ記述子は、リストの最初または最後の項目でなければなりません。

Example

例

テストグループ A、B、C、D、E を含むテストスイートについて、次の例のリストに、IDT が最初にテストグループ A を実行し、次にテストグループ B を実行し、次いで C、D、E を任意の順序で実行するよう指定するためのさまざまな方法を示します。

-
```
Order:
    - - A
      - B
      - [C, D, E]
```

- ```
  Order:
      - - A
        - B
        - "*"
  ```
- ```
  Order:
      - - A
        - B

      - - B
        - C

      - - B
        - D

      - - B
        - E
  ```

Features

オプション。IDT に awsiotdevicetester_report.xml ファイルに追加させる製品機能のリスト。このセクションを指定しない場合、IDT はレポートに製品機能を追加しません。

製品機能とは、デバイスが満たしている可能性のある特定の基準に関するユーザー定義の情報です。例えば、MQTT 製品機能には、デバイスが MQTT メッセージを適切に公開することを指定できます。awsiotdevicetester_report.xml では、製品機能は指定されたテストが合格したかどうかに応じて、supported、not-supported、またはカスタムのユーザー定義値に設定されます。

Features リストの各項目は以下のパラメータで設定されます。

Name

機能の名前。

Value

オプション。supported の代わりにレポートで使用するカスタム値。この値を指定しない場合、テスト結果に基づいて、IDT により機能値が supported または not-supported に設定されます。同じ機能を異なる条件でテストする場合、Features リストにおけるその機能のインスタンスごとにカスタム値を使用できます。IDT は、サポート条件の機能値を連結します。詳細については、以下を参照してください。

`Condition`

ブール値に評価されるコンテキスト式。評価値が true の場合、IDT はテストスイートを実行後、その機能をテストレポートに追加します。評価値が false の場合、テストはレポートに含まれません。

`Tests`

オプション。テスト記述子のリスト。機能をサポートするには、このリストで指定されるテストにすべて合格する必要があります。

このリストの各テスト記述子は、テストグループ ID と 1 つ以上のテストケース ID を使用して、特定のテストグループから実行する個々のテストを識別します。テスト記述子は以下の形式を使用します。

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

`Features` リストの各機能について、`Tests` か `OneOfTests` のどちらかを指定する必要があります。

`OneOfTests`

オプション。テスト記述子のリスト。機能をサポートするには、このリストで指定されているテストのうち少なくとも 1 つに合格する必要があります。

このリストの各テスト記述子は、テストグループ ID と 1 つ以上のテストケース ID を使用して、特定のテストグループから実行する個々のテストを識別します。テスト記述子は以下の形式を使用します。

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

`Features` リストの各機能について、`Tests` か `OneOfTests` のどちらかを指定する必要があります。

`IsRequired`

機能がテストレポートに必要かどうかを定義するブール値。デフォルト値は `false` です。

テストオーケストレーターコンテキスト

テストオーケストレーターコンテキストは、実行中のテストオーケストレーターに利用可能なデータが含まれている読み取り専用 JSON ドキュメントです。テストオーケストレーターコンテキストは、テストオーケストレーターからのみアクセス可能で、テストフローを決定する情報が含まれています。例えば、テストの実行者によって userdata.json ファイルに設定された情報を使用して、特定のテストを実行する必要があるかどうかを決定できます。

テストオーケストレーターコンテキストは次の形式を使用します。

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    }
}
```

pool

テスト実行用に選択されたデバイスプールに関する情報。選択されたデバイスプールのこの情報は、device.json ファイルで定義された、対応する最上位レベルのデバイスプール配列要素から取得されます。

userData

userdata.json ファイル内の情報。

config

config.json ファイル内の情報。

コンテキストは、JSONPath 表記法を使用してクエリできます。ステート定義における JSonPath クエリの構文は {{*query*}} です。テストオーケストレーターコンテキストからデータにアクセスする場合、各値が文字列、数値、またはブール値として評価されることを確認してください。

JSONPath 表記を使用してコンテキストのデータにアクセスする方法の詳細については、IDT コンテキストを使用するを参照してください。

IDT ステートマシンを設定する

> **Important**
>
> IDT v4.5.2 以降、このステートマシンは非推奨です。新しいテストオーケストレーターを使用することを強くお勧めします。詳細については、「IDT テストオーケストレーターを設定する」を参照してください。

ステートマシンは、テストスイートの実行フローを制御するコンストラクトです。テストスイートの開始ステートを決定し、ユーザー定義のルールに基づいてステートの移行を管理し、終了ステートに達するまでステートの移行を継続します。

テストスイートにユーザー定義のステートマシンが含まれていない場合は、IDT によってステートマシンが生成されます。デフォルトのステートマシンには、次の機能があります。

- テストの実行者に、テストスイート全体ではなく、特定のテストグループを選択して実行する機能を提供する。
- 特定のテストグループが選択されていない場合、テストスイート内のすべてのテストグループをランダムな順序で実行する。
- レポートを生成し、各テストグループおよびテストケースのテスト結果を示すコンソールサマリーを出力する。

IDT テストスイートのステートマシンは、次の基準を満たす必要があります。

- 各ステートが、IDT が実行する各アクション (テストグループの実行、レポートファイルの生成など) に対応する。
- ステートが移行すると、そのステートに関連付けられたアクションを実行する。
- 各ステートが、次のステートの移行ルールを定義する。
- 終了ステートが `Succeed` または `Fail` である。

ステートマシンの形式

次のテンプレートを使用して、独自の *<custom-test-suite-folder>*`/suite/state_machine.json` ファイルを設定できます。

```
{
```

```
  "Comment": "<description>",
  "StartAt": "<state-name>",
  "States": {
    "<state-name>": {
      "Type": "<state-type>",
      // Additional state configuration
    }

    // Required states
    "Succeed": {
      "Type": "Succeed"
    },
    "Fail": {
      "Type": "Fail"
    }
  }
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Comment

ステートマシンの説明。

StartAt

IDT がテストスイートの実行を開始するステートの名前。StartAt の値は、States オブジェクトにリストされているいずれかのステートに設定する必要があります。

States

ユーザー定義のステート名を有効な IDT ステートにマッピングするオブジェクト。各 States.*state-name* オブジェクトには、*state-name* にマッピングされた有効なステートの定義が含まれています。

States オブジェクトには、Succeed ステートおよび Fail ステートを含める必要があります。有効なステートについては、有効なステートとステートの定義を参照してください。

有効なステートとステートの定義

このセクションでは、IDT ステートマシンで使用可能なすべての有効なステートのステート定義について説明します。以下に示すステートの一部は、テストケースレベルでの設定をサポートしていま

す。ただし、絶対に必要な場合を除き、テストケースレベルではなく、テストグループレベルでステート移行ルールを設定することをお勧めします。

ステートの定義

- RunTask
- 選択
- 並行
- AddProductFeatures
- レポートを行う
- LogMessage
- SelectGroup
- 失敗
- 成功

RunTask

RunTask ステートは、テストスイートで定義されているテストグループからテストケースを実行します。

```
{
    "Type": "RunTask",
    "Next": "<state-name>",
    "TestGroup": "<group-id>",
    "TestCases": [
        "<test-id>"
    ],
    "ResultVar": "<result-name>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Next`

現在のステートのアクションを実行した後に移行するステートの名前。

`TestGroup`

オプション。実行するテストグループの ID。この値を指定しない場合、IDT はテストの実行者が選択するテストグループを実行します。

TestCases

オプション。TestGroup に指定されたグループのテストケース ID の配列。IDT は、TestGroup と TestCases の値に基づいて、次のようにテストの実行動作を決定します。

- TestGroup と TestCases 両方が指定されている場合、IDT はテストグループから指定されたテストケースを実行します。
- TestCases が指定され、TestGroup が指定されていない場合、IDT は指定されたテストケースを実行します。
- TestGroup が指定され、TestCases が指定されていない場合は、IDT は指定されたテストグループ内のすべてのテストケースを実行します。
- TestGroup も TestCases も指定されていない場合、IDT は、テストの実行者が IDT CLI から選択したテストグループからすべてのテストケースを実行します。テストの実行者がグループを選択できるようにするには、statemachine.json ファイルに RunTask ステートと Choiceステート両方を含める必要があります。これを行う方法の例については、ステートマシンの例: ユーザーが選択したテストグループを実行するを参照してください。

  テストの実行者向けの IDT CLI コマンドを有効にする方法については「the section called “IDT CLI コマンドを有効にする”」を参照してください。

ResultVar

テスト実行の結果によって設定するコンテキスト変数の名前。TestGroup の値を指定しなかった場合は、この値を指定しないでください。IDT は、以下に基づいて、ResultVar に定義された変数を true または false に設定します。

- 変数名の形式が *text*_*text*_passed の場合、この値は、最初のテストグループのすべてのテストが合格したか、スキップされたかに設定されます。
- それ以外の場合、この値は、すべてのテストグループのすべてのテストが合格したか、スキップされたかに設定されます。

通常、RunTask ステートは、個々のテストケース ID を指定せずにテストグループ ID を指定するために使用されます。この指定により、IDT は指定されたテストグループ内のすべてのテストケースを実行します。このステートで実行されるすべてのテストケースは、ランダムな順序で並行して実行されます。ただし、すべてのテストケースが実行に 1 つのデバイスを必要とし、単一のデバイスしか使用できない場合は、テストケースは順次実行されます。

エラー処理

指定されたテストグループ ID またはテストケース ID のいずれかが有効でない場合、このステートは RunTaskError 実行エラーを発行します。またこのステートは、実行エラーに遭遇すると、ステートマシンコンテキスト内の hasExecutionError 変数を true に設定します。

選択

Choice ステートでは、ユーザー定義の条件に基づいて、移行先の次のステートを動的に設定できます。

```
{
    "Type": "Choice",
    "Default": "<state-name>",
    "FallthroughOnError": true | false,
    "Choices": [
        {
            "Expression": "<expression>",
            "Next": "<state-name>"
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Default

Choices に定義されているいずれの式も true に評価されない場合に移行先になるデフォルトのステート。

FallthroughOnError

オプション。このステートが式評価エラーに遭遇したときの動作を指定します。評価結果がエラーになったときに式をスキップしたい場合は true に設定します。一致する式がない場合、ステートマシンは Default ステートに移行します。FallthroughOnError 値は、指定されない場合、デフォルトで false になります。

Choices

現在のステートのアクションを実行した後に移行するステートを決定する式とステートの配列。

Choices.Expression

ブール値に評価される式文字列。式が true と評価された場合、ステートマシンは Choices.Next に定義されているステートに移行します。式文字列は、ステートマシンコン

テキストから値を取得し、オペレーションを実行してブール値に到達します。ステートマシンコンテキストへのアクセスの詳細については、「ステートマシンコンテキスト」を参照してください。

`Choices.Next`

`Choices.Expression` で定義されている式が `true` に評価された場合の移行先のステート名。

エラー処理

以下に示すケースでは、Choice ステートでエラー処理が必要になることがあります。

- choice 式の一部の変数が、ステートマシンのコンテキストに存在しない。
- 式の結果がブール値ではない。
- JSON 検索の結果が、文字列、数値、またはブール値ではない。

このステートのエラー処理に `Catch` ブロックを使用することはできません。ステートマシンがエラーに遭遇したときに、その実行を停止するには、`FallthroughOnError` を `false` に設定する必要があります。ただし、`FallthroughOnError` は `true` に設定し、ユースケースに応じて、次のいずれかの操作を実行することをお勧めします。

- アクセスしている変数が一部のケースに存在しないと考えられる場合は、`Default` の値と追加の `Choices` ブロックを使用して次のステートを指定します。
- 使用している変数が必ず存在するものである場合は、`Default` ステートを `Fail` に設定します。

並行

`Parallel` ステートでは、新しいステートマシンを互いに並列に定義して実行できます。

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Branches": [
        <state-machine-definition>
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

Branches

実行するステートマシン定義の配列。各ステートマシン定義には、それぞれの StartAt、Succeed、および Fail ステートを含める必要があります。この配列内のステートマシン定義は、各自の定義外のステートを参照することはできません。

> **Note**
>
> 各ブランチステートマシンは同じステートマシンコンテキストを共有するため、あるブランチに変数を設定し、別のブランチからそれらの変数を読み込むと、予期しない動作が発生する可能性があります。

Parallel ステートは、すべてのブランチステートマシンを実行してから次のステートに移行します。デバイスを必要とする各ステートは、デバイスが利用可能になるまで実行を待ちます。複数のデバイスが利用可能な場合、このステートは並行して複数のグループからテストケースを実行します。十分な数のデバイスが利用できない場合、テストケースは順次実行されます。テストケースは、並列して実行される場合、ランダムな順序で実行されるため、同じテストグループからのテストの実行に異なるデバイスが使用されることがあります。

エラー処理

実行エラーを処理するには、ブランチステートマシンと親ステートマシンの両方が、Fail ステートに移行していることを確認します。

ブランチステートマシンは親ステートマシンに実行エラーを送信しないため、ブランチステートマシンの実行エラーを処理するために Catch ブロックを使用することはできません。代わりに、共有ステートマシンコンテキストの hasExecutionErrors 値を使用します。これを行う方法の例については、ステートマシンの例: 2 つのテストグループを並行して実行するを参照してください。

AddProductFeatures

AddProductFeatures ステートでは、IDT によって生成される awsiotdevicetester_report.xml ファイルに製品機能を追加できます。

製品機能とは、デバイスが満たしている可能性のある特定の基準に関するユーザー定義の情報です。例えば、MQTT 製品機能には、デバイスが MQTT メッセージを適切に公開することを指定できま

す。レポートでは、製品機能は指定されたテストが合格したかどうかに応じて、supported、not-supported、カスタム値に設定されます。

> **Note**
>
> AddProductFeatures ステートだけではレポートは生成されません。レポートを生成するには、このステートが Report ステートに移行する必要があります。

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Features": [
        {
            "Feature": "<feature-name>",
            "Groups": [
                "<group-id>"
            ],
            "OneOfGroups": [
                "<group-id>"
            ],
            "TestCases": [
                "<test-id>"
            ],
            "IsRequired": true | false,
            "ExecutionMethods": [
                "<execution-method>"
            ]
        }
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

Features

awsiotdevicetester_report.xml ファイルに表示される製品機能の配列。

Feature

機能の名前。

FeatureValue

オプション。supported の代わりにレポートで使用するカスタム値。この値を指定しない場合、テスト結果に基づいて、機能値は supported または not-supported に設定されます。

FeatureValue にカスタム値を使用する場合は、同じ機能を異なる条件でテストできます。IDT は、サポート条件の機能値を連結します。例えば、以下の抜粋は、MyFeature 機能と 2 つの異なる機能値を示しています。

```
...
{
    "Feature": "MyFeature",
    "FeatureValue": "first-feature-supported",
    "Groups": ["first-feature-group"]
},
{
    "Feature": "MyFeature",
    "FeatureValue": "second-feature-supported",
    "Groups": ["second-feature-group"]
},
...
```

両方のテストグループが合格した場合、機能値は first-feature-supported, second-feature-supported に設定されます。

Groups

オプション。テストグループ ID の配列。機能をサポートするには、指定された各テストグループ内のすべてのテストが合格である必要があります。

OneOfGroups

オプション。テストグループ ID の配列。機能をサポートするには、指定されたテストグループうち、少なくとも 1 つのグループに含まれるすべてのテストが合格である必要があります。

TestCases

オプション。テストケース ID の配列。この値を指定すると、次のことが適用されます。

- 機能をサポートするには、指定されたすべてのテストケースが合格である必要があります。
- Groups には、テストグループ ID を 1 つだけ含める必要があります。
- OneOfGroups は指定できません。

IsRequired

オプション。この機能をレポートでオプション機能としてマークするには、false に設定します。デフォルト値は true です。

ExecutionMethods

オプション。device.json ファイルに指定された protocol 値と一致する実行メソッドの配列。この値を指定した場合、この機能をレポートに含めるには、テストの実行者はこの配列の値の 1 つに一致する protocol 値を指定する必要があります。この値を指定しない場合、この機能は常にレポートに含まれます。

AddProductFeatures ステートを使用するには、RunTask ステートの ResultVar の値を以下のいずれかの値に指定する必要があります。

- 個々のテストケース ID を指定した場合は、ResultVar を*group-id_test-id*_passed に指定します。
- 個々のテストケース ID を指定しなかった場合は、ResultVar を*group-id*_passed に指定します。

AddProductFeatures ステートは、次の方法でテスト結果をチェックします。

- テストケース ID を指定しなかった場合は、各テストグループの結果は、ステートマシンコンテキスト内の *group-id*_passed 変数の値から決定されます。
- テストケース ID を指定した場合は、各テストの結果は、ステートマシンコンテキスト内の *group-id_test-id*_passed 変数の値から決定されます。

エラー処理

このステートで指定されたグループ ID が有効なグループ ID でない場合、このステートで AddProductFeaturesError 実行エラーが発生します。またこのステートは、実行エラーに遭遇すると、ステートマシンコンテキスト内の hasExecutionErrors 変数を true に設定します。

レポートを行う

Report ステートでは、*suite-name*_Report.xml ファイルと awsiotdevicetester_report.xml ファイルが生成されます。またこのステートでは、レポートがコンソールにストリーミングされます。

```
{
    "Type": "Report",
    "Next": "<state-name>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

テストの実行者がテスト結果を確認できるように、テスト実行フローの終了ステートの前に Report ステートに移行する必要があります。通常、このステートの次のステートは Succeed です。

エラー処理

このステートは、レポート生成時に問題に遭遇した場合、ReportError 実行エラーを発行します。

LogMessage

LogMessage ステートでは、test_manager.log ファイルが生成され、ログメッセージがコンソールにストリーミングされます。

```
{
    "Type": "LogMessage",
    "Next": "<state-name>"
    "Level": "info | warn | error"
    "Message": "<message>"
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

Level

ログメッセージを作成するエラーレベル。有効でないレベルを指定すると、エラーメッセージが生成され、そのレベルは破棄されます。

Message

ログに記録するメッセージ。

SelectGroup

SelectGroup ステートでは、ステートマシンコンテキストを更新して選択されたグループを示します。このステートで設定した値は、後続のすべての Choice ステートによって使用されます。

```
{
    "Type": "SelectGroup",
    "Next": "<state-name>"
    "TestGroups": [
        <group-id>"
    ]
}
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

Next

現在のステートのアクションを実行した後に移行するステートの名前。

TestGroups

選択済みとしてマークされるテストグループの配列。この配列の各テストグループ ID について、*group-id*_selected 変数がコンテキストで true に設定されます。IDT は、指定されたグループが存在するかどうかを検証しないため、有効なテストグループ ID を指定するようにしてください。

失敗

Fail ステートは、ステートマシンが正しく実行されなかったことを示します。これはステートマシンの終了ステートです。各ステートマシンの定義にこのステートを含める必要があります。

```
{
```

```
    "Type": "Fail"
}
```

成功

Succeed ステートは、ステートマシンが正しく実行されたことを示します。これはステートマシンの終了ステートです。各ステートマシンの定義にこのステートを含める必要があります。

```
{
    "Type": "Succeed"
}
```

ステートマシンコンテキスト

ステートマシンコンテキストは、実行中のステートマシンに利用可能なデータが含まれている読み取り専用 JSON ドキュメントです。ステートマシンコンテキストは、ステートマシンからのみアクセス可能で、テストフローを決定する情報が含まれています。例えば、テストの実行者によって userdata.json ファイルに設定された情報を使用して、特定のテストを実行する必要があるかどうかを決定できます。

ステートマシンコンテキストでは、次の形式が使用されます。

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    },
    "suiteFailed": true | false,
    "specificTestGroups": [
        "<group-id>"
    ],
    "specificTestCases": [
        "<test-id>"
    ],
    "hasExecutionErrors": true
}
```

pool

テスト実行用に選択されたデバイスプールに関する情報。選択されたデバイスプールのこの情報は、device.json ファイルで定義された、対応する最上位レベルのデバイスプール配列要素から取得されます。

userData

userdata.json ファイル内の情報。

config

config.json ファイル内の情報。

suiteFailed

この値は、ステートマシンが起動すると false に設定されます。テストグループが RunTask ステートで失敗した場合、この値はステートマシン実行の残りの時間の間 true に設定されます。

specificTestGroups

テストの実行者がテストスイート全体ではなく特定のテストグループを選択して実行する場合に、このキーが作成され、特定のテストグループ ID のリストが格納されます。

specificTestCases

テストの実行者がテストスイート全体ではなく特定のテストケースを選択して実行する場合に、このキーが作成され、特定のテストケース ID のリストが格納されます。

hasExecutionErrors

ステートマシンの起動時には存在しません。いずれかのステートが実行エラーに遭遇した場合に、この変数が作成され、ステートマシンの実行の残りの時間の間 true に設定されます。

コンテキストは、JSONPath 表記法を使用してクエリできます。ステート定義における JSonPath クエリの構文は {{$.*query*}} です。JSONPath クエリは、一部のステートではプレースホルダー文字列として使用できます。IDT は、プレースホルダー文字列をコンテキストから評価された JSONPath クエリの値に置き換えます。プレースホルダーは、次の値に使用できます。

- RunTask ステートの TestCases 値。
- Choice ステートの Expression 値。

ステートマシンコンテキストからデータにアクセスする場合は、次の条件を満たしていることを確認します。

- JSON パスが `$.` で始まっている。
- 各値が、文字列、数値、またはブール値として評価される。

JSONPath 表記を使用してコンテキストのデータにアクセスする方法の詳細については、IDT コンテキストを使用するを参照してください。

実行エラー

実行エラーとは、ステートの実行時にステートマシンが遭遇する、ステートマシン定義内のエラーです。IDT は、各エラーに関する情報を `test_manager.log` ファイルに記録し、ログメッセージをコンソールにストリーミングします。

実行エラーは、次の方法を使用して処理できます。

- ステート定義内に `Catch`ブロック を追加する。
- ステートマシンコンテキストで `hasExecutionErrors` 値 の値を確認する。

Catch

`Catch` を使用するには、ステート定義に以下を追加します。

```
"Catch": [
    {
        "ErrorEquals": [
            "<error-type>"
        ]
        "Next": "<state-name>"
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

`Catch.ErrorEquals`

キャッチするエラータイプの配列。実行エラーが指定された値のいずれかと一致する場合、ステートマシンは、`Catch.Next` に指定されているステートに移行します。生成されるエラーのタイプの詳細については、各ステート定義を参照してください。

Catch.Next

現在のステートが、Catch.ErrorEquals に指定されている値のいずれかと一致する実行エラーに遭遇した場合に、移行する次のステート。

キャッチブロックは、いずれかが一致するまで順番に処理されます。どのエラーもキャッチブロックに指定されているエラーと一致しない場合、ステートマシンは実行を継続します。実行エラーは誤ったステート定義によって発生するため、ステートが実行エラーに遭遇したときは Fail ステートに移行することをお勧めします。

hasExecutionError

一部のステートは、実行エラーに遭遇した場合、エラーを発行するだけでなく、ステートマシンコンテキストの hasExecutionError 値も true に設定します。この値を使用して、エラーがいつ発生したかを特定してから、Choice ステートを使用してステートマシンを Fail ステートに移行することができます。

この方法には次の特徴があります。

- ステートマシンは、hasExecutionError に値が割り当てられていると開始しません。またこの値は、特定のステートによって設定されるまで得られません。つまり、明示的に FallthroughOnError の値を false に設定することによって、実行エラーが発生していない場合に、この値にアクセスする Choice ステートがステートマシンを停止しないようにする必要があります。
- hasExecutionError は、一度 true に設定されると、false に設定されることも、コンテキストから削除されることもありません。つまり、この値は true に設定された初回のみ有効であり、以降のすべてのステートに対して意味のある値を提供しないことを意味します。
- hasExecutionError 値は Parallel ステート内のすべてのブランチステートマシンで共有されるため、アクセスされる順序によっては、予期せぬ結果が発生する可能性があります。

これらの特性から、代わりに Catch ブロックを使用できる場合は、この方法を使用することはお勧めしません。

ステートマシンの例

このセクションでは、ステートマシンの設定の例を紹介します。

例

- ステートマシンの例: 1 つのテストグループを実行する

- ステートマシンの例: ユーザーが選択したテストグループを実行する
- ステートマシンの例: 製品機能が含まれる 1 つのテストグループを実行する
- ステートマシンの例: 2 つのテストグループを並行して実行する

ステートマシンの例: 1 つのテストグループを実行する

このステートマシンの動作:

- ID GroupA のテストグループを実行します。このテストグループは、group.json ファイルのスイート内に存在している必要があります。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

```
{
    "Comment": "Runs a single group and then generates a report.",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
```

```
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: ユーザーが選択したテストグループを実行する

このステートマシンの動作:

- テストの実行者が特定のテストグループを選択したかどうかをチェックします。テストの実行者がテストケースを選択するには、テストグループも選択する必要があるため、ステートマシンは特定のテストケースはチェックしません。
- テストグループが選択されている場合:
  - 選択されたテストグループ内のテストケースを実行します。そのために、ステートマシンは、RunTask ステートでは、テストグループまたはテストケースを明示的に指定しません。
  - すべてのテストを実行した後にレポートを生成し、終了します。
- テストグループが選択されていない場合:
  - テストグループ GroupA のテストを実行します。
  - レポートを生成して終了します。

```
{
    "Comment": "Runs specific groups if the test runner chose to do that, otherwise
 runs GroupA.",
    "StartAt": "SpecificGroupsCheck",
    "States": {
        "SpecificGroupsCheck": {
            "Type": "Choice",
            "Default": "RunGroupA",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.specificTestGroups[0]}} != ''",
```

```
                "Next": "RunSpecificGroups"
            }
        ]
    },
    "RunSpecificGroups": {
        "Type": "RunTask",
        "Next": "Report",
        "Catch": [
            {
                "ErrorEquals": [
                    "RunTaskError"
                ],
                "Next": "Fail"
            }
        ]
    },
    "RunGroupA": {
        "Type": "RunTask",
        "Next": "Report",
        "TestGroup": "GroupA",
        "Catch": [
            {
                "ErrorEquals": [
                    "RunTaskError"
                ],
                "Next": "Fail"
            }
        ]
    },
    "Report": {
        "Type": "Report",
        "Next": "Succeed",
        "Catch": [
            {
                "ErrorEquals": [
                    "ReportError"
                ],
                "Next": "Fail"
            }
        ]
    },
    "Succeed": {
        "Type": "Succeed"
    },
```

```
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: 製品機能が含まれる 1 つのテストグループを実行する

このステートマシンの動作:

- テストグループ GroupA を実行します。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。
- FeatureThatDependsOnGroupA 機能を awsiotdevicetester_report.xml ファイルに追加します。
  - GroupA が合格である場合、機能を supported に設定します。
  - レポートでこの機能をオプションとしてマークしません。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

```
{
    "Comment": "Runs GroupA and adds product features based on GroupA",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "AddProductFeatures",
            "TestGroup": "GroupA",
            "ResultVar": "GroupA_passed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
```

```
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

ステートマシンの例: 2 つのテストグループを並行して実行する

このステートマシンの動作:

- GroupA および GroupB テストグループを並行して実行します。ブランチステートマシンの RunTask ステートによってコンテキストに格納された ResultVar 変数が AddProductFeatures ステートに利用可能になります。
- 実行エラーをチェックし、エラーが見つかった場合は Fail に移行します。このステートマシンは、Catch ブロックを使用しません。この方法では、ブランチステートマシンの実行エラーが検出されないためです。

- 合格したグループに基づいて、awsiotdevicetester_report.xml ファイルに機能を追加します。
  - GroupA が合格である場合、機能を supported に設定します。
  - レポートでこの機能をオプションとしてマークしません。
- エラーがない場合には、レポートを生成し、Succeed に移行します。エラーがある場合は、Fail に移行します。

デバイスプールに 2 つのデバイスが設定されている場合、GroupA と GroupB 両方を同時に実行できます。ただし、GroupA または GroupB のどちらかに複数のテストが含まれている場合は、両方のデバイスがそれらのテストに割り当てられることがあります。デバイスが 1 つだけ設定されている場合、テストグループは順次実行されます。

```
{
    "Comment": "Runs GroupA and GroupB in parallel",
    "StartAt": "RunGroupAAndB",
    "States": {
        "RunGroupAAndB": {
            "Type": "Parallel",
            "Next": "CheckForErrors",
            "Branches": [
                {
                    "Comment": "Run GroupA state machine",
                    "StartAt": "RunGroupA",
                    "States": {
                        "RunGroupA": {
                            "Type": "RunTask",
                            "Next": "Succeed",
                            "TestGroup": "GroupA",
                            "ResultVar": "GroupA_passed",
                            "Catch": [
                                {
                                    "ErrorEquals": [
                                        "RunTaskError"
                                    ],
                                    "Next": "Fail"
                                }
                            ]
                        },
                        "Succeed": {
                            "Type": "Succeed"
```

```
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            },
            {
                "Comment": "Run GroupB state machine",
                "StartAt": "RunGroupB",
                "States": {
                    "RunGroupA": {
                        "Type": "RunTask",
                        "Next": "Succeed",
                        "TestGroup": "GroupB",
                        "ResultVar": "GroupB_passed",
                        "Catch": [
                            {
                                "ErrorEquals": [
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            }
        ]
    },
    "CheckForErrors": {
        "Type": "Choice",
        "Default": "AddProductFeatures",
        "FallthroughOnError": true,
        "Choices": [
            {
                "Expression": "{{$.hasExecutionErrors}} == true",
                "Next": "Fail"
            }
        ]
```

```
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                },
                {
                    "Feature": "FeatureThatDependsOnGroupB",
                    "Groups": [
                        "GroupB"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

IDT テストケース実行可能ファイルを作成する

テストケース実行可能ファイルは、次の方法でテストスイートフォルダ内に作成して配置できます。

- `test.json` ファイル内の引数または環境変数を使用して実行するテストを決定するテストスイートの場合は、テストスイート全体に対して 1 つのテストケース実行可能ファイルを作成することも、テストスイート内のテストグループごとに 1 つのテスト実行可能ファイルを作成することもできます。
- 指定したコマンドに基づいて特定のテストを実行するテストスイートの場合は、テストスイート内のテストケースごとに 1 つのテストケース実行可能ファイルを作成します。

テストを作成するユーザーは、ユースケースに適したアプローチを決定し、それに応じてテストケースの実行可能ファイルを構成できます。各 `test.json` ファイルに、正しいテストケース実行可能ファイルのパスを指定していること、および指定した実行可能ファイルが正常に実行されることを確認してください。

すべてのデバイスにテストケースを実行する準備が整うと、IDT は以下のファイルを読み取ります。

- `test.json`。選択されたテストケースが、開始するプロセスと設定する環境変数を決定するために使用します。
- `suite.json`。テストスイートが、設定する環境変数を決定するために使用します。

IDT は、`test.json` ファイルで指定されているコマンドと引数に基づいて、必要なテスト実行可能プロセスを開始し、必要な環境変数をプロセスに渡します。

IDT クライアント SDK を使用する

IDT クライアント SDK を使用すると、IDT とテスト対象のデバイスとのやり取りに使用できる API コマンドを使用して、テスト実行可能ファイルにテストロジックを簡単に記述できます。現在、IDT では次の SDK が用意されています。

- IDT クライアント SDK for Python
- IDT クライアント SDK for Go
- IDT Client SDK for Java

これらの SDK は、*<device-tester-extract-location>*/sdks フォルダにあります。新しいテストケース実行可能ファイルを作成するときは、使用する SDK をテストケース実行可能ファイ

ルが含まれるフォルダにコピーし、コード内で SDK を参照する必要があります。このセクションでは、テストケースの実行可能ファイルで使用できる API コマンドについて簡単に説明します。

このセクションの内容

- デバイスとのやり取り
- IDT とのやり取り
- ホストとのやり取り

デバイスとのやり取り

次のコマンドを使用すると、デバイスとのやり取りや接続管理のための追加の関数を実装せずに、テスト対象デバイスと通信することができます。

`ExecuteOnDevice`

テストスイートが、SSH または Docker シェル接続をサポートするデバイス上で、シェルコマンドを実行できるようにします。

`CopyToDevice`

テストスイートが、IDT を実行するホストマシンから、SSH または Docker シェル接続をサポートするデバイス上の指定された場所にローカルファイルをコピーできるようにします。

`ReadFromDevice`

テストスイートが、UART 接続をサポートするデバイスのシリアルポートから読み取りできるようにします。

> **Note**
>
> IDT は、コンテキストからのデバイスアクセス情報を使用して確立されたデバイスへの直接接続を管理しないため、テストケース実行可能ファイルでは、デバイスとやり取り用のこれらの API コマンドを使用することをお勧めします。ただし、これらのコマンドがテストケースの要件を満たしていない場合は、IDT コンテキストからデバイスアクセス情報を取得し、この情報を使用してテストスイートからデバイスに直接接続できます。
> 直接接続するには、テスト対象デバイスとリソースデバイスそれぞれの `device.connectivity` フィールドと `resource.devices.connectivity` フィールドの情報を取得します。IDT コンテキスト使用の詳細については、IDT コンテキストを使用するを参照してください。

IDT とのやり取り

次のコマンドを使用すると、テストスイートが IDT と通信できるようになります。

`PollForNotifications`

テストスイートが IDT からの通知をチェックできるようにします。

`GetContextValue` および **`GetContextString`**

テストスイートが IDT コンテキストから値を取得できるようにします。詳細については、IDT コンテキストを使用する を参照してください。

`SendResult`

テストスイートがテストケースの結果を IDT にレポートできるようにします。このコマンドは、テストスイートの各テストケースの最後に呼び出す必要があります。

ホストとのやり取り

次のコマンドを使用すると、テストスイートがホストマシンと通信できるようになります。

`PollForNotifications`

テストスイートが IDT からの通知をチェックできるようにします。

`GetContextValue` および **`GetContextString`**

テストスイートが IDT コンテキストから値を取得できるようにします。詳細については、IDT コンテキストを使用する を参照してください。

`ExecuteOnHost`

テストスイートがローカルマシン上でコマンドを実行できるようにし、IDT がテストケース実行可能ファイルのライフサイクルを管理できるようにします。

IDT CLI コマンドを有効にする

`run-suite` コマンド IDT CLI には、テストの実行者がテスト実行をカスタマイズするためのいくつかのオプションがあります。テストの実行者がこれらのオプションを使用してカスタムテストスイートを実行できるようにするには、IDT CLI のサポートを実装します。サポートを実装しなくてもテストは実行できますが、一部の CLI オプションは正しく機能しません。理想的なカスタマーエクスペ

リエンスを提供するために、IDT CLI で `run-suite` コマンドの次の引数のサポートを実装することをお勧めします。

`timeout-multiplier`

テストの実行中にすべてのタイムアウトに適用される 1.0 より大きい値を指定します。

テストの実行者は、この引数を使用して、実行するテストケースのタイムアウトを増やすことができます。テストの実行者が `run-suite` コマンドにこの引数を指定すると、IDT はこの値を使用して IDT_TEST_TIMEOUT 環境変数の値を計算し、IDT コンテキストの `config.timeoutMultiplier` フィールドを設定します。この引数をサポートするには、以下の手順を実行する必要があります。

- `test.json` ファイルのタイムアウト値を直接使用する代わりに、IDT_TEST_TIMEOUT 環境変数を読み取り、正しく計算されたタイムアウト値を取得します。
- IDT コンテキストから `config.timeoutMultiplier` 値を取得し、長時間実行されるタイムアウトに適用します。

タイムアウトイベントによる早期終了の詳細については、終了動作を指定するを参照してください。

`stop-on-first-failure`

障害が発生した場合に、IDT がすべてのテスト実行を停止するように指定します。

テストの実行者がこの引数を `run-suite` コマンドを指定すると、IDT は障害が発生するとすぐにテストの実行を停止します。ただし、テストケースが並行して実行されている場合、この設定によって予期しない結果につながる可能性があります。このサポートを実装するには、テストロジックを使用して、IDT がこのイベントに遭遇した場合に、実行中のすべてのテストケースに対して、実行を停止し、一時リソースをクリーンアップし、テスト結果を IDT にレポートするように指示します。障害発生時の早期終了の詳細については、終了動作を指定するを参照してください。

`group-id` および **`test-id`**

IDT が選択されたテストグループまたはテストケースのみを実行するように指定します。

テストの実行者は、`run-suite` コマンドでこれらの引数を使用して、以下のテスト実行可能ファイルの動作を指定できます。

- 指定されたテストスイート内のすべてのテストグループを実行する。
- 指定されたテストグループ内から選択したテストを実行する。

これらの引数をサポートするには、テストスイート用のステートマシンに、自分のステートマシンの RunTask ステートおよび Choice ステートのセットが含まれている必要があります。カスタムステートマシンを使用しない場合は、デフォルトの IDT ステートマシンに必要なステートが含まれているため、追加のアクションを行う必要はありません。ただし、カスタムステートマシンを使用している場合は、サンプルとして ステートマシンの例: ユーザーが選択したテストグループを実行する を使用して、自分のステートマシンに必要なステートを追加してください。

IDT CLI コマンドの詳細については、カスタムテストスイートのデバッグと実行を参照してください。

イベントログの書き込み

テストの実行中は、イベントログとエラーメッセージをコンソールに書き込むために stdout と stderr にデータを送信します。コンソールメッセージの形式の詳細については、コンソールメッセージの形式を参照してください。

IDT がテストスイートの実行を終了すると、この情報は *<devicetester-extract-location>*/results/*<execution-id>*/logs フォルダにある test_manager.log ファイルでも利用可能になります。

各テストケースは、テスト実行のログ (テスト対象デバイスのログを含む) を *<device-tester-extract-location>*/results/*execution-id*/logs フォルダにある *<group-id>*_*<test-id>* ファイルに書き込むように設定できます。これを行うには、testData.logFilePath クエリを使用して IDT コンテキストからログファイルへのパスを取得し、そのパスにファイルを作成し、必要なコンテンツをそのファイルに書き込みます。IDT は、実行中のテストケースに基づいてこのパスを自動的に更新します。テストケースのログファイルを作成しないことを選択すると、そのテストケースのファイルは生成されません。

また、必要に応じて *<device-tester-extract-location>*/logs フォルダに追加のログファイルを作成するようにテキスト実行可能ファイルをセットアップすることもできます。ファイルが上書きされないように、ログファイル名に一意のプレフィックスを指定することをお勧めします。

IDT に結果をレポートする

IDT は、テスト結果を awsiotdevicetester_report.xml ファイルと *suite-name*_report.xml ファイルに書き込みます。これらのレポートファイルは、*<device-tester-extract-location>*/results/*<execution-id>*/ にあります。両レポートとも、テストスイート実行の結果をキャプチャします。IDT がこれらのレポートに使用するスキーマの詳細については、IDT テストの結果とログを確認するを参照してください。

*suite-name*_report.xml ファイルのコンテンツを取得するには、SendResult コマンドを使用して、テスト実行が終了する前に、テスト結果を IDT にレポートする必要があります。IDT は、テスト結果を見つけられない場合、テストケースのエラーを発行します。次の Python の抜粋は、テスト結果を IDT に送信するコマンドを示しています。

```
request-variable = SendResultRequest(TestResult(result))
client.send_result(request-variable)
```

API を使用して結果をレポートしない場合、IDT はテストアーティファクトフォルダでテスト結果を検索します。このフォルダのパスは、IDT コンテキストの testData.testArtifactsPath フィールドに格納されています。このフォルダで、IDT は、アルファベット順にソートされた最初の XML ファイルをテスト結果として使用します。

テストロジックが JUnit XML 結果を生成する場合は、結果を解析してから API を使用して IDT に送信する代わりに、アーティファクトフォルダ内の XML ファイルにテスト結果を書き込んで、直接 IDT に提供することができます。

この方法を使用する場合は、テストロジックによってテスト結果が正確に要約されていること、および *suite-name*_report.xml ファイルと同じ形式で結果ファイルがフォーマットされていることを確認してください。IDT は、次の例外を除き、提供されたデータの検証を実行しません。

- IDT は testsuites タグのすべてのプロパティを無視します。代わりに、レポートされた他のテストグループ結果からタグのプロパティを計算します。
- testsuite 内に少なくとも 1 つの testsuites タグが存在する必要があります。

IDT はすべてのテストケースで同じアーティファクトフォルダを使用し、テスト実行の終了後、次のテスト実行までに結果ファイルを削除しないため、この方法を使用すると、IDT が正しくないファイルを読み取った場合に、誤ったレポートが行われる可能性もあります。IDT が適切な結果を読み取れるように、すべてのテストケースで生成される XML 結果ファイルに同じ名前を使用して、各テストケースの結果を上書きすることをお勧めします。テストスイートのレポート作成に複合的なアプローチ (一部のテストケースには XML 結果ファイルを使用し、他のテストケースには API を使用して結果を送信する) を使用することもできますが、このアプローチはお勧めしません。

終了動作を指定する

テキスト実行可能ファイルは、テストケースが障害やエラー結果をレポートした場合でも、常に終了コード 0 で終了するように設定します。ゼロ以外の終了コードは、テストケースが実行されなかったこと、またはテストケース実行可能ファイルが結果を IDT に通知できなかったことを示す場合に

のみ使用します。IDT は、0 以外の終了コードを受信すると、テスト実行を妨げるエラーが発生したとしてテストケースをマークします。

IDT は、以下に示すイベントが発生すると、終了前にテストケースに実行の停止を要求 (または想定) することがあります。以下の情報を使用して、テストケースから以下の各イベントを検出するようにテストケース実行可能ファイルを設定します。

タイムアウト

テストケースが、test.json ファイルで指定されたタイムアウト値よりも長く実行されたときに発生します。テストの実行者が timeout-multiplier 引数を使用してタイムアウト乗数を指定すると、IDT はこの乗数を使用してタイムアウト値を計算します。

このイベントを検出するには、IDT_TEST_TIMEOUT 環境変数を使用します。テストの実行者がテストを起動すると、IDT は IDT_TEST_TIMEOUT 環境変数の値を計算されたタイムアウト値 (秒単位) に設定し、その変数をテストケース実行可能ファイルに渡します。この変数の値を読み取って適切なタイマーを設定します。

割り込み

テストの実行者が IDT に割り込むと発生します。例えば、Ctrl+C を押した時です。

ターミナルはすべての子プロセスに通知を伝播するため、割り込み通知を検出する通知ハンドラをテストケースに簡単に設定できます。

または、API を定期的にポーリングして、PollForNotifications API 応答の CancellationRequested ブール値をチェックできます。IDT は割り込み通知を受信すると、CancellationRequested ブールの値を true に設定します。

最初の失敗時に停止する

現在のテストケースと並行して実行中のテストケースが失敗し、テストの実行者が stop-on-first-failure 引数を使用して、障害の発生時に IDT が実行を停止するように設定しているときに発生します。

このイベントを検出するには、PollForNotifications API を定期的にポーリングして、API レスポンスの CancellationRequested ブールの値をチェックします。IDT は、最初の障害発生時に停止するように設定されている場合、障害に遭遇すると、CancellationRequested ブールの値を true に設定します。

これらのいずれかのイベントが発生すると、IDT は現在実行中のテストケースの実行が終了するまで 5 分間待機します。実行中のすべてのテストケースが 5 分以内に終了しない場合、IDT は各プロセス

を強制的に停止させます。IDT は、プロセスの終了前にテスト結果を受け取らなかった場合、テストケースをタイムアウトしたとしてマークします。ベストプラクティスとして、いずれかのイベントが発生したときは、テストケースが以下のアクションを実行するようにします。

1. 通常のテストロジックの実行を停止する。
2. テスト対象デバイスのテストアーティファクトなど、すべての一時的なリソースをクリーンアップする。
3. テスト結果 (テストの失敗やエラーなど) を IDT にレポートする。
4. 終了する。

IDT コンテキストを使用する

IDT がテストスイートを実行するとき、テストスイートは、各テストの実行方法の決定に使用できる一連のデータにアクセスできます。このデータは IDT コンテキストと呼ばれます。例えば、テストの実行者によって `userdata.json` ファイルに提供されるユーザーデータ設定は、IDT コンテキスト内でテストスイートに提供されます。

IDT コンテキストは、読み取り専用の JSON ドキュメントと考えることができます。テストスイートは、オブジェクト、配列、数値などの標準 JSON データ型を使用して、コンテキストからデータを取得することや、コンテキストにデータを書き込むことができます。

コンテキストスキーマ

IDT コンテキストは次の形式を使用します。

```
{
    "config": {
        <config-json-content>
        "timeoutMultiplier": timeout-multiplier,
        "idtRootPath": <path/to/IDT/root>
    },
    "device": {
        <device-json-device-element>
    },
    "devicePool": {
        <device-json-pool-element>
    },
    "resource": {
        "devices": [
```

```
            {
                <resource-json-device-element>
                "name": "<resource-name>"
            }
        ]
    },
    "testData": {
        "awsCredentials": {
            "awsAccessKeyId": "<access-key-id>",
            "awsSecretAccessKey": "<secret-access-key>",
            "awsSessionToken": "<session-token>"
        },
        "logFilePath": "/path/to/log/file"
    },
    "userData": {
        <userdata-json-content>
    }
}
```

config

config.json ファイル からの情報。config フィールドには、次の追加フィールドも含まれます。

config.timeoutMultiplier

テストスイートによって使用される任意のタイムアウト値の乗数。この値は、IDT CLI からテストの実行者によって指定されます。デフォルト値は 1 です。

config.idRootPath

この値は、userdata.json ファイルを設定する際の IDT の絶対パス値のプレースホルダーです。この値はビルドコマンドとフラッシュコマンドで使用されます。

device

テスト実行用に選択されたデバイスに関する情報。この情報は、選択されたデバイスの device.json ファイル の devices 配列要素に相当します。

devicePool

テスト実行用に選択されたデバイスプールに関する情報。この情報は、選択されたデバイスプールの device.json ファイルに定義されている最上位レベルのデバイスプール配列要素に相当します。

resource

resource.json ファイルからのリソースデバイスに関する情報。

resource.devices

この情報は、devices ファイルに定義されている resource.json 配列に相当します。各 devices 要素には、以下の追加フィールドが含まれています。

resource.device.name

リソースデバイスの名前。この値は、test.json ファイルで requiredResource.name 値に設定されます。

testData.awsCredentials

クラウドに接続 AWS するためにテストで使用される AWS 認証情報。この情報は、config.json ファイルから取得されます。

testData.logFilePath

テストケースがログメッセージを書き込むログファイルへのパス。このファイルは、存在しない場合、テストスイートによって作成されます。

userData

テストの実行者によって userdata.json ファイル に提供された情報。

コンテキスト内のデータにアクセスする

コンテキストは、JSONPath 表記を使用して設定ファイルからクエリすることも、GetContextValue および GetContextString API を使用してテキスト実行可能ファイルからクエリすることもできます。IDT コンテキストにアクセスするための JSONPath 文字列の構文は、次のように異なります。

- suite.json および test.json では、{{*query*}} を使用します。つまり、式を開始するためにルート要素 $. を使用しません。
- statemachine.json では {{$.*query*}} を使用します。
- API コマンドでは、コマンドに応じて *query* または {{$.*query*}} を使用します。詳細については、SDK のインラインドキュメントを参照してください。

次の表に、一般的な foobar JSONPath 式の演算子を示します。

演算子	説明
$	ルート要素。IDT の最上位レベルのコンテキスト値はオブジェクトであるため、通常は $. を使用してクエリを開始します。
.childName	オブジェクトから、名前 childName を使用して子要素にアクセスします。配列に適用されると、この演算子が各要素に適用された新しい配列が生成されます。この要素名は、大文字と小文字が区別されます。例えば、config オブジェクトの awsRegion 値にアクセスするクエリは $.config.awsRegion です。
[start:end]	配列から要素をフィルターし、start インデックスから end インデックスまでの項目を取得します (ともに境界値を含む)。
[index1, index2, ... , indexN]	配列から要素をフィルターし、指定されたインデックスのみから項目を取得します。
[?(expr)]	expr 式を使用して配列から要素をフィルターします。この式は、ブール値に評価される必要があります。

フィルター式を作成するには、次の構文を使用します。

```
<jsonpath> | <value> operator <jsonpath> | <value>
```

この構文の説明は次のとおりです。

- jsonpath は、標準 JSON 構文を使用する JSONPath です。
- value は、標準 JSON 構文を使用するカスタム値です。
- operator は、以下のいずれかの演算子です。
  - < (未満)
  - <= (以下)

- == (等しい)

  式内の JSONPath または値が配列、ブール値、またはオブジェクト値である場合は、これがユーザーに使用可能な唯一の二項演算子です。
- >= (以上)
- > (次より大きい)
- =~ (正規表現の一致)。この演算子をフィルター式で使用するには、式の左側の JSONPath または値が文字列に評価される必要があり、右側が RE2 構文 に従ったパターン値である必要があります。

{{*query*}} 形式の JSONPath クエリは、プレースホルダ文字列として、`test.json` ファイルの `args` および `environmentVariables` フィールド内と、`suite.json` ファイルの `environmentVariables` フィールド内で使用できます。IDT はコンテキスト検索を実行し、クエリの評価値をフィールドに入力します。例えば、`suite.json` ファイルでは、プレースホルダー文字列を使用して、各テストケースとともに変化する環境変数の値を指定できます。IDT は、環境変数に各テストケースの正しい値を入力します。ただし、`test.json` ファイルおよび `suite.json` ファイルでプレースホルダー文字列を使用する場合は、クエリに次の考慮事項が適用されます。

- クエリに含まれる各 `devicePool` キーは、すべて小文字にする必要があります。つまり、代わりに `devicepool` を使用します。
- 配列には、文字列の配列のみを使用できます。さらに、配列は非標準の `item1,item2,...,itemN` の形式を使用します。配列は、要素が 1 つしか含まれていない場合、`item` としてシリアル化され、文字列フィールドと区別がつかなくなります。
- プレースホルダーを使用してコンテキストからオブジェクトを取得することはできません。

これらの事項を考慮して、テストロジックのコンテキストへのアクセスには、`test.json` ファイルおよび `suite.json` ファイルのプレースホルダー文字列ではなく、可能な限り API を使用することをお勧めします。ただし、環境変数として設定する単一の文字列を取得するときは、JSONPath プレースホルダーを使用する方が便利な場合があります。

テストの実行者向けの設定の構成

カスタムテストスイートを実行するには、テストの実行者は、実行するテストスイートに基づいて設定を設定する必要があります。設定は、*<device-tester-extract-location>*/configs/ フォルダにある設定ファイルテンプレートに基づいて指定します。必要に応じて、テストランナーは IDT が AWS クラウドへの接続に使用する AWS 認証情報も設定する必要があります。

テストを作成するユーザーは、テストスイートをデバッグするために、以下に示すファイルの設定が必要になります。また、テストスイートを実行するために必要な以下の設定を設定できるように、テストの実行者に指示を提供する必要があります。

device.json の設定

device.json ファイルには、テストが実行されるデバイスに関する情報 (IP アドレス、ログイン情報、オペレーティングシステム、CPU アーキテクチャなど) が含まれています。

テストの実行者は、*<device-tester-extract-location>*/configs/ フォルダにある次のテンプレート device.json ファイルを使用してこの情報を指定できます。

```
[
    {
        "id": "<pool-id>",
        "sku": "<pool-sku>",
        "features": [
            {
                "name": "<feature-name>",
                "value": "<feature-value>",
                "configs": [
                    {
                        "name": "<config-name>",
                        "value": "<config-value>"
                    }
                ],
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "pairedResource": "<device-id>", //used for no-op protocol
                "connectivity": {
                    "protocol": "ssh | uart | docker | no-op",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "publicKeyPath": "<public-key-path>",
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
```

```
                        "privKeyPath": "/path/to/private/key",

                        // password
                        "password": "<password>",
                    }
                },

                // uart
                "serialPort": "<serial-port>",

                // docker
                "containerId": "<container-id>",
                "containerUser": "<container-user-name>",
            }
        }
    ]
}
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

id

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

sku

テスト対象デバイスを一意に識別する英数字の値。SKU は、認定されたデバイスの追跡に使用されます。

> **Note**
>
> AWS Partner Device Catalog にボードを一覧表示する場合、ここで指定する SKU は、一覧表示プロセスで使用する SKU と一致する必要があります。

features

オプション。デバイスでサポートされている機能を含む配列。デバイス機能は、テストスイートに設定するユーザー定義の値です。テストの実行者に、device.json ファイルに含める機

能名および値に関する情報を提供する必要があります。例えば、他のデバイスの MQTT サーバーとして機能するデバイスをテストする場合は、MQTT_QoS という名前の機能に対する特定のサポートレベルを検証するようにテストロジックを設定します。テストの実行者は、この機能名を指定し、デバイスによってサポートされる QoS レベルにその機能値を設定します。指定された情報は、devicePool.features クエリを使用して IDT コンテキストから、または pool.features クエリを使用してステートマシンコンテキストから取得できます。

features.name

機能の名前。

features.value

サポートされている機能値。

features.configs

機能の構成設定 (必要な場合)。

features.config.name

構成設定の名前。

features.config.value

サポートされている設定値。

devices

テスト対象のプール内のデバイスの配列。少なくとも 1 つのデバイスが必要です。

devices.id

テスト対象のデバイスのユーザー定義の一意の識別子。

devices.pairedResource

リソースデバイスに対してユーザーが定義した一意の識別子。この値は、no-op 接続プロトコルを使用してデバイスをテストする場合に必要です。

connectivity.protocol

このデバイスと通信するために使用される通信プロトコル。プール内の各デバイスは、同じプロトコルを使用する必要があります。

現在、サポートされている値は、物理デバイスの場合は ssh と uart、Docker コンテナの場合は docker、IDT ホストマシンと直接接続されていないが、ホストマシンと通信するための物理ミドルウェアとしてリソースデバイスを必要とするデバイスの場合は no-op だけです。

no-op デバイスの場合は、devices.pairedResource でリソースデバイス ID を設定します。この ID も resource.json ファイルで指定する必要があります。ペアリングするデバイスは、テスト対象のデバイスと物理的にペアリングされているデバイスである必要があります。IDT は、ペアリングされたリソースデバイスを識別して接続した後は、test.json ファイルに記述されている機能に従って他のリソースデバイスに接続することはありません。

connectivity.ip

テスト対象のデバイスの IP アドレス。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.port

オプション。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.publicKeyPath

オプション。テスト対象のデバイスへの接続を認証するために使用される公開キーへのフルパス。publicKeyPath を指定すると、IDT がテスト対象のデバイスへの SSH 接続を確立するときに、デバイスの公開キーを検証します。この値が指定されていない場合、IDT は SSH 接続を作成しますが、デバイスのパブリックキーは検証しません。

公開キーへのパスを指定し、安全な方法を使用して、この公開キーをフェッチすることを強くお勧めします。標準のコマンドラインベースの SSH クライアントの場合、パブリックキーは known_hosts ファイルで提供されます。別の公開キーファイルを指定する場合、このファイルは known_hosts ファイルと同じ形式、つまり (ip-address key-type public-key) を使用する必要があります。

connectivity.auth

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.password

テスト中のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

connectivity.auth.credentials.privKeyPath

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

connectivity.auth.credentials.user

テスト対象デバイスにサインインするためのユーザー名。

connectivity.serialPort

オプション。デバイスが接続されているシリアルポート。

このプロパティは、connectivity.protocol が uart に設定されている場合にのみ適用されます。

connectivity.containerId

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、connectivity.protocol が docker に設定されている場合にのみ適用されます。

connectivity.containerUser

オプション。コンテナ内のユーザー名。デフォルト値は Dockerfile で指定されたユーザーです。

デフォルト値は 22 です。

このプロパティは、connectivity.protocol が docker に設定されている場合にのみ適用されます。

> **Note**
>
> テストの実行者がテストに対して誤ったデバイス接続を構成しているかどうかを確認するには、ステートマシンコンテキストから pool.Devices[0].Connectivity.Protocol を取得し、この値を Choice ステート内の予想値と比較します。正しくないプロトコルが使用されている場合は、LogMessage ステートを使用してメッセージを出力し、Fail ステートに移行します。
> または、エラー処理コードを使用して、誤ったデバイスタイプによるテスト失敗をレポートすることもできます。

(オプション) userdata.json の設定

userdata.json ファイルには、device.json ファイルには指定されていない、テストスイートに必要とされる追加情報が含まれています。このファイルの形式は、テストスイートに定義されている userdata_scheme.json ファイルによって異なります。テストを作成するユーザーは、作成したテストスイートを実行するユーザーにこの情報を提供してください。

(オプション) resource.json の設定

resource.json ファイルには、リソースデバイスとして使用されるすべてのデバイスに関する情報が含まれています。リソースデバイスは、テスト対象のデバイスの特定の機能をテストするために必要なデバイスです。例えば、デバイスの Bluetooth 機能をテストするには、リソースデバイスを使用して、デバイスがリソースデバイスに正常に接続できるかどうかをテストできます。リソースデバイスはオプションで、必要な数だけリソースデバイスを要求できます。テストを作成するユーザーは、test.json ファイル を使用して、テストに必要なリソースデバイスの機能を定義します。テストの実行者は、resource.json ファイルを使用して、必要な機能を持つリソースデバイスのプールを指定します。作成したテストスイートを実行するユーザーに、以下の情報を提供してください。

テストの実行者は、*<device-tester-extract-location>*/configs/ フォルダにある次のテンプレート resource.json ファイルを使用してこの情報を指定できます。

```
[
    {
```

```
        "id": "<pool-id>",
        "features": [
            {
                "name": "<feature-name>",
                "version": "<feature-value>",
                "jobSlots": <job-slots>
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "publicKeyPath": "<public-key-path>",
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

id

デバイスプールと呼ばれるデバイスのコレクションを一意に識別するユーザー定義の英数字の ID。プールに属するデバイスには、同一のハードウェアが必要です。テストスイートを実行する場合、プールのデバイスを使用してワークロードを並列化します。複数のデバイスを使用して異なるテストを実行します。

features

オプション。デバイスでサポートされている機能を含む配列。このフィールドに必要な情報は、テストスイートの test.json ファイル に定義されています。この情報によって、実行するテストと、テストの実行方法が決まります。テストスイートに機能が必要ない場合は、このフィールドは必須ではありません。

features.name

機能の名前。

features.version

機能バージョン。

features.jobSlots

デバイスを同時に使用できるテストの数を示すための設定。デフォルト値は 1 です。

devices

テスト対象のプール内のデバイスの配列。少なくとも 1 つのデバイスが必要です。

devices.id

テスト対象のデバイスのユーザー定義の一意の識別子。

connectivity.protocol

このデバイスと通信するために使用される通信プロトコル。プール内の各デバイスは、同じプロトコルを使用する必要があります。

現在、サポートされている値は、物理デバイス用の `ssh` および `uart` と、Docker コンテナ用の `docker` のみです。

connectivity.ip

テスト対象のデバイスの IP アドレス。

このプロパティは、`connectivity.protocol` が `ssh` に設定されている場合にのみ適用されます。

connectivity.port

オプション。SSH 接続に使用するポート番号。

デフォルト値は 22 です。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.publicKeyPath

オプション。テスト対象のデバイスへの接続を認証するために使用される公開キーへのフルパス。publicKeyPath を指定すると、IDT がテスト対象のデバイスへの SSH 接続を確立するときに、デバイスの公開キーを検証します。この値が指定されていない場合、IDT は SSH 接続を作成しますが、デバイスのパブリックキーは検証しません。

公開キーへのパスを指定し、安全な方法を使用して、この公開キーをフェッチすることを強くお勧めします。標準のコマンドラインベースの SSH クライアントの場合、パブリックキーは known_hosts ファイルで提供されます。別の公開キーファイルを指定する場合、このファイルは known_hosts ファイルと同じ形式、つまり (ip-address key-type public-key) を使用する必要があります。

connectivity.auth

接続の認証情報。

このプロパティは、connectivity.protocol が ssh に設定されている場合にのみ適用されます。

connectivity.auth.method

指定された接続プロトコルを介してデバイスにアクセスするために使用される認証方法。

サポートされている値は以下のとおりです。

- pki
- password

connectivity.auth.credentials

認証に使用される認証情報。

connectivity.auth.credentials.password

テスト中のデバイスにサインインするためのパスワード。

この値は、connectivity.auth.method が password に設定されている場合にのみ適用されます。

connectivity.auth.credentials.privKeyPath

テスト中のデバイスにサインインするためのプライベートキーへの完全パス。

この値は、connectivity.auth.method が pki に設定されている場合にのみ適用されます。

connectivity.auth.credentials.user

テスト対象デバイスにサインインするためのユーザー名。

connectivity.serialPort

オプション。デバイスが接続されているシリアルポート。

このプロパティは、connectivity.protocol が uart に設定されている場合にのみ適用されます。

connectivity.containerId

テスト対象の Docker コンテナのコンテナ ID または名前。

このプロパティは、connectivity.protocol が docker に設定されている場合にのみ適用されます。

connectivity.containerUser

オプション。コンテナ内のユーザー名。デフォルト値は Dockerfile で指定されたユーザーです。

デフォルト値は 22 です。

このプロパティは、connectivity.protocol が docker に設定されている場合にのみ適用されます。

(オプション) config.json の設定

config.json ファイルには、IDT 向けの設定情報が含まれています。通常、テストランナーは、IDT、およびオプションで AWS リージョンの AWS ユーザー認証情報を提供する場合を除き、このファイルを変更する必要はありません。必要なアクセス許可を持つ認証情報が提供されている場合、 AWS は使用状況メトリクスを AWS IoT Device Tester 収集して送信します AWS。これはオプ

トイン機能で、IDT 機能を改善するために使用されます。詳細については、「IDT 使用状況メトリクスを送信する」を参照してください。

テストランナーは、次のいずれかの方法で AWS 認証情報を設定できます。

- 認証情報ファイル

  IDT では、 AWS CLIと同じ認証情報ファイルが使用されます。詳細については、「設定ファイルと認証情報ファイル」を参照してください。

  認証情報ファイルの場所は、使用しているオペレーティングシステムによって異なります。

  - macOS、Linux: `~/.aws/credentials`
  - Windows: `C:\Users\UserName\.aws\credentials`
- 環境変数

  環境変数は、オペレーティングシステムによって維持され、システムコマンドによって使用される変数です。SSH セッション中に定義された変数は、そのセッションの終了後は使用できません。IDT は、環境変数の `AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY` を使用して AWS 認証情報を保存します。

  これらの変数を Linux、macOS、または Unix で設定するには、export を使用します。

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

  Windows でこれらの変数を設定するには、set を使用します。

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

IDT の AWS 認証情報を設定するには、テストランナーは *<device-tester-extract-location>*/configs/フォルダにある config.json ファイルの authセクションを編集します。

```
{
    "log": {
        "location": "logs"
    },
    "configFiles": {
        "root": "configs",
```

```
        "device": "configs/device.json"
    },
    "testPath": "tests",
    "reportPath": "results",
    "awsRegion": "<region>",
    "auth": {
        "method": "file | environment",
        "credentials": {
            "profile": "<profile-name>"
        }
    }
}
]
```

以下に説明するように、値が含まれているすべてのフィールドは必須です。

> **Note**
>
> このファイル内のすべてのパスは、*<device-tester-extract-location>* に関連して定義されています。

log.location

<device-tester-extract-location> のログフォルダへのパス。

configFiles.root

設定ファイルが含まれるフォルダへのパス。

configFiles.device

device.json ファイルへのパス。

testPath

テストスイートが含まれるフォルダへのパス。

reportPath

IDT がテストスイートを実行した後にテスト結果が含まれるフォルダへのパス。

awsRegion

オプション。テストスイートが使用する AWS リージョン。設定されない場合、テストスイートは各テストスイートに指定されているデフォルトのリージョンを使用します。

`auth.method`

IDT が AWS 認証情報を取得するために使用するメソッド。サポートされる値は、`file` (認証情報ファイルから認証情報を取得) と `environment` (環境変数を使用して認証情報を取得) です。

`auth.credentials.profile`

認証情報ファイルから使用する認証情報プロファイル。このプロパティは、`auth.method` が `file` に設定されている場合にのみ適用されます。

カスタムテストスイートのデバッグと実行

必要な設定 を終了すると、IDT はテストスイートを実行することができます。完全なテストスイートの実行時間は、ハードウェアとテストスイートの構成によって異なります。参考までに、Raspberry Pi 3B で完全な FreeRTOS 認定テストスイートを完了するには約 30 分かかります。

テストスイートの作成中に、IDT を使用してテストスイートをデバッグモードで実行すると、テストスイートを実行する前やテストの実行者に提供する前に、コードをチェックすることができます。

IDT をデバッグモードで実行する

テストスイートは、IDT に依存してデバイスとやり取りし、コンテキストを提供し、結果を受け取るため、IDT と通信しないと、IDE でテストスイートを簡単にデバッグすることはできません。そのために、IDT CLI は IDT をデバッグモードで実行できるようにする `debug-test-suite` コマンドを提供します。`debug-test-suite` で使用可能なオプションを表示するには、次のコマンドを実行します。

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

IDT をデバッグモードで実行するとき、IDT は実際にテストスイートを起動したり、テストオーケストレーターを実行したりしません。代わりに、IDE と対話して IDE で実行されているテストスイートによるリクエストに応答して、コンソールにログを出力します。IDT はタイムアウトせず、手動で中断されるまで待機してから終了します。デバッグモードでは、IDT はテストオーケストレーターも実行せず、レポートファイルも生成しません。テストスイートをデバッグするには、通常は IDT が設定ファイルから取得する情報を、IDE を使用して提供する必要があります。以下の情報を提供してください。

- 各テストの環境変数と引数。IDT はこの情報を `test.json` または `suite.json` から読み込みません。
- リソースデバイスを選択するための引数。IDT はこの情報を `test.json` から読み込みません。

テストスイートをデバッグするには、次の手順を実行します。

1. テストスイートの実行に必要な構成設定ファイルを作成します。例えば、テストスイートが device.json、resource.json、および user data.json を必要とする場合は、必要に応じてこれらすべてを設定してください。
2. 次のコマンドを実行して IDT をデバッグモードにし、テストの実行に必要なデバイスを選択します。

   ```
   devicetester_[linux | mac | win_x86-64] debug-test-suite [options]
   ```

   このコマンドを実行すると、IDT はテストスイートからのリクエストを待機し、それらのリクエストに応答します。IDT は、IDT クライアント SDK がケースを処理するために必要な環境変数も生成します。
3. IDE で、run または debug 設定を使用して次の手順を実行します。

   a. IDT で生成された環境変数の値を設定します。

   b. test.json ファイルと suite.json ファイルに指定したすべての環境変数または引数の値を設定します。

   c. 必要に応じてブレークポイントを設定します。
4. IDE でテストスイートを実行します。

   テストスイートは、必要に応じて何度でもデバッグして再実行できます。IDT はデバッグモードではタイムアウトしません。
5. デバッグが完了したら、IDT を中断してデバッグモードを終了します。

テストを実行する IDT CLI コマンド

次のセクションでは、IDT CLI コマンドについて説明します。

IDT v4.0.0

help

指定されたコマンドに関する情報を一覧表示します。

list-groups

特定のテストスイート内のグループを一覧表示します。

list-suites

使用可能なテストスイートを一覧表示します。

list-supported-products

お使いの IDT バージョン (この場合は FreeRTOS バージョン) のサポート対象製品と、現在の IDT バージョンで利用可能な FreeRTOS 認定テストスイートのバージョンを一覧表示します。

list-test-cases

指定したテストグループのテストケースを一覧表示します。次のオプションがサポートされています。

- group-id。検索するテストグループ。このオプションは必須で、1 つのグループを指定する必要があります。

run-suite

デバイスプールに対してテストスイートを実行します。以下に、一般的に使用されるオプションの一部を示します。

- suite-id。実行するテストスイートのバージョン。指定しない場合、IDT は tests フォルダにある最新バージョンを使用します。
- group-id。実行するテストグループ (カンマ区切りリストとして)。指定しない場合、IDT はテストスイートのすべてのテストグループを実行します。
- test-id。実行するテストケース (カンマ区切りリストとして)。指定した場合は、group-id は 1 つのグループを指定する必要があります。
- pool-id。テストするデバイスプール。device.json ファイルに複数のデバイスプールが定義されている場合、テストの実行者は 1 つのプールを指定する必要があります。
- timeout-multiplier。テスト用の test.json ファイルに指定されているテスト実行タイムアウトを、ユーザー定義乗数を使用して変更するように IDT を設定します。
- stop-on-first-failure。最初に障害が発生した時点で実行を停止するように IDT を設定します。指定されたテストグループをデバッグするには、このオプションを group-id とともに使用する必要があります。
- userdata。テストスイートの実行に必要なユーザーデータ情報を含むファイルを設定します。テストスイートの suite.json ファイルで、userdataRequired が true に設定されている場合にのみ必要です。

run-suite オプションの詳細については、次の help オプションを使用してください。

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

debug-test-suite

デバッグモードでテストスイートを実行します。詳細については、IDT をデバッグモードで実行する を参照してください。

IDT テストの結果とログを確認する

このセクションでは、IDT がコンソールログとテストレポートを生成する形式について説明します。

コンソールメッセージの形式

AWS IoT Device Tester は、テストスイートの開始時にコンソールにメッセージを印刷するための標準形式を使用します。以下の抜粋は、IDT によって生成されるコンソールメッセージの例を示しています。

```
[INFO] [2000-01-02 03:04:05]: Using suite: MyTestSuite_1.0.0
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

コンソールメッセージの大半は、次のフィールドで設定されます。

time

ログに記録されたイベントの完全な ISO 8601 タイムスタンプ。

level

ログに記録されたイベントのメッセージレベル。通常、ログに記録されるメッセージレベルは、`info`、`warn`、または `error` のいずれかです。IDT は、早期終了の原因となる予期されるイベントが発生した場合は、`fatal` または `panic` メッセージを発行します。

msg

ログに記録されたメッセージ。

executionId

現在の IDT プロセスの一意の ID 文字列。この ID は、個々の IDT 実行を区別するために使用されます。

テストスイートから生成されたコンソールメッセージは、テスト対象のデバイスとテストスイート、テストグループ、IDT が実行するテストケースに関する追加情報を提供します。次の抜粋は、テストスイートから生成されたコンソールメッセージの例を示しています。

```
[INFO] [2000-01-02 03:04:05]: Hello world! suiteId=MyTestSuitegroupId=myTestGroup
 testCaseId=myTestCase deviceId=my-
deviceexecutionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

コンソールメッセージのテストスイート固有の部分には、次のフィールドが含まれています。

suiteId

現在実行中のテストスイートの名前。

groupId

現在実行中のテストグループの ID。

testCaseId

現在実行中のテストケースの ID。

deviceId

現在のテストケースが使用しているテスト対象デバイスの ID。

テストサマリーには、テストスイート、実行された各グループのテスト結果、生成されたログファイルとレポートファイルの場所に関する情報が含まれています。次の例は、テストサマリーメッセージを示しています。

```
========== Test Summary ==========
Execution Time:     5m00s
Tests Completed:    4
Tests Passed:       3
Tests Failed:       1
Tests Skipped:      0
---------------------------------
Test Groups:
    GroupA:         PASSED
    GroupB:         FAILED
---------------------------------
Failed Tests:
    Group Name: GroupB
        Test Name: TestB1
```

```
            Reason: Something bad happened
---------------------------------
Path to AWS IoT Device Tester Report: /path/to/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/logs
Path to Aggregated JUnit Report: /path/to/MyTestSuite_Report.xml
```

AWS IoT Device Tester レポートスキーマ

`awsiotdevicetester_report.xml` は、次の情報が含まれる署名済みレポートです。

- IDT バージョン。
- テストスイートのバージョン。
- レポートの署名に使用されるレポートの署名とキー。
- `device.json` ファイルで指定されているデバイス SKU とデバイスプール。
- テストされた製品のバージョンとデバイスの機能。
- テスト結果の概要の集計。この情報は、*suite-name*`_report.xml` ファイルに含まれている情報と同じです。

```
<apnreport>
    <awsiotdevicetesterversion>idt-version</awsiotdevicetesterversion>
    <testsuiteversion>test-suite-version</testsuiteversion>
    <signature>signature</signature>
    <keyname>keyname</keyname>
    <session>
        <testsession>execution-id</testsession>
        <starttime>start-time</starttime>
        <endtime>end-time</endtime>
    </session>
    <awsproduct>
        <name>product-name</name>
        <version>product-version</version>
        <features>
            <feature name="<feature-name>" value="supported | not-supported | <feature-
value>" type="optional | required"/>
        </features>
    </awsproduct>
    <device>
        <sku>device-sku</sku>
        <name>device-name</name>
        <features>
```

```
            <feature name="<feature-name>" value="<feature-value>"/>
        </features>
        <executionMethod>ssh | uart | docker</executionMethod>
    </device>
    <devenvironment>
        <os name="<os-name>"/>
    </devenvironment>
    <report>
        <suite-name-report-contents>
    </report>
</apnreport>
```

awsiotdevicetester_report.xml ファイルには、テスト対象の製品および一連のテストの実行後に検証された製品機能に関する情報を含む <awsproduct> タグが含まれています。

<awsproduct> タグで使用される属性

name

テスト対象の製品の名前。

version

テスト対象の製品のバージョン。

features

検証された機能です。required としてマークされている機能は、テストスイートがデバイスを検証するために必要です。次のスニペットは、この情報が awsiotdevicetester_report.xml ファイルで表示される方法を示します。

```
<feature name="ssh" value="supported" type="required"></feature>
```

optional としてマークされている機能は、検証に必須ではありません。次のスニペットは、オプションの機能を示しています。

```
<feature name="hsi" value="supported" type="optional"></feature>
<feature name="mqtt" value="not-supported" type="optional"></feature>
```

テストスイートのレポートスキーマ

*suite-name*_Result.xml レポートは JUnit XML 形式です。Jenkins、Bamboo などのように継続的な統合 (CI) と継続的なデプロイ (CD) のプラットフォームに統合することができます。このレポートには、テスト結果の概要の集計が含まれています。

```
<testsuites name="<suite-name> results" time="<run-duration>" tests="<number-of-test>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
    <testsuite name="<test-group-id>" package="" tests="<number-of-tests>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
        <!--success-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>"/>
        <!--failure-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <failure type="<failure-type>">
                reason
            </failure>
        </testcase>
        <!--skipped-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <skipped>
                reason
            </skipped>
        </testcase>
        <!--error-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <error>
                reason
            </error>
        </testcase>
    </testsuite>
</testsuites>
```

awsiotdevicetester_report.xml と *suite-name*_report.xml 両方のレポートセクションには、実行されたテストとその結果が一覧表示されます。

最初の XML タグ <testsuites> には、テストの実行の概要が含まれています。例:

```
<testsuites name="MyTestSuite results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

<testsuites> タグで使用される属性

name

テストスイートの名前。

time

スイートの実行所要時間 (秒)。

tests

実行されたテストの数。

failures

実行されたテストのうち、合格しなかったものの数。

errors

IDT で実行できなかったテストの数。

disabled

この属性は使用されていないため無視できます。

テストに障害やエラーが発生した場合は、<testsuites> XML タグを確認することで、障害の生じたテストを特定できます。<testsuites> タグ内の <testsuite> XML タグは、テストグループのテスト結果の要約を示します。例:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

形式は <testsuites> タグと似ていますが、使用されていないため無視できる skipped という属性があります。各 <testsuite> XML タグ内には、テストグループの実行されたテスト別の <testcase> タグがあります。例:

```
<testcase classname="Security Test" name="IP Change Tests" attempts="1"></testcase>
```

<testcase> タグで使用される属性

name

テストの名前。

attempts

IDT でテストケースを実行した回数。

テストに障害やエラーが発生した場合、<failure> タグまたは <error> タグがトラブルシューティングのための情報とともに <testcase> タグに追加されます。例:

```
<testcase classname="mcu.Full_MQTT" name="MQTT_TestCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
</testcase>
```

IDT 使用状況メトリクスを送信する

必要なアクセス許可を持つ AWS 認証情報を提供すると、 は使用状況メトリクスを AWS IoT Device Tester 収集して送信します AWS。これはオプトイン機能で、IDT 機能を改善するために使用されます。IDT は次のような情報を収集します。

- IDT の実行に使用される AWS アカウント ID
- テストの実行に使用される IDT CLI コマンド
- 実行されるテストスイート
- *<device-tester-extract-location>* フォルダにあるテストスイート
- デバイスプール内に設定されているデバイスの数
- テストケース名と実行時間
- テストに合格したか、失敗したか、エラーが発生したか、スキップされたかなどのテスト結果情報
- テストされた製品の機能
- 予期せぬ終了、早期終了などの IDT 終了動作

IDT が送信するすべての情報は、*<device-tester-extract-location>*/results/*<execution-id>*/ フォルダの metrics.log ファイルにもログが記録されます。ログファイルを表示すると、テスト実行中に収集された情報を確認できます。このファイルは、使用状況メトリックを収集することを選択した場合にのみ生成されます。

メトリクスの収集を無効にするために、追加のアクションを実行する必要はありません。単に AWS 認証情報を保存しないでください。 AWS 認証情報を保存している場合は、それらにアクセスするように config.json ファイルを設定しないでください。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、AWS IAM Identity Center 「 ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン 「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。

- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。

- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

IDT に AWS 認証情報を提供する

IDT が AWS 認証情報にアクセスしてメトリクスを送信できるようにするには AWS、以下を実行します。

1. IAM ユーザーの AWS 認証情報を環境変数として、または認証情報ファイルに保存します。

   a. 環境変数を使用するには、次のコマンドを実行します。

   ```
   AWS_ACCESS_KEY_ID=access-key
   AWS_SECRET_ACCESS_KEY=secret-access-key
   ```

   b. 認証情報ファイルを使用するには、`.aws/credentials file:` に次の情報を追加します。

   ```
   [profile-name]
   aws_access_key_id=access-key
   aws_secret_access_key=secret-access-key
   ```

2. `config.json` ファイルの `auth` セクションを設定します。詳細については、「(オプション) config.json の設定」を参照してください。

テストスイートのバージョンを維持する

IDT for FreeRTOS は、テストリソースをテストスイートとテストグループに整理します。

- テストスイートは、デバイスが FreeRTOS の特定のバージョンで動作することを確認するために使用されるテストグループのセットです。
- テストグループは、BLE や MQTT メッセージングなど、特定の機能に関連する個々のテストのセットです。

IDT v3.0.0 以降、テストスイートは、1.0.0 から始まる `major.minor.patch` 形式でバージョン管理されます。IDT をダウンロードすると、パッケージに最新のテストスイートのバージョンが含まれます。

コマンドラインインターフェイスで IDT を起動すると、IDT は新しいテストスイートのバージョンが使用可能かどうかをチェックします。使用可能である場合は、新しいバージョンに更新するよう求められます。現在のテストを更新するか、続行するかを選択できます。

> **Note**
>
> IDT では、認定のために 3 つの最新のテストスイートバージョンをサポートしています。詳細については、のサポートポリシーを理解する AWS IoT Device Tester を参照してください。

`upgrade-test-suite` コマンドを使用して、テストスイートをダウンロードできます。または、IDT の起動時にオプションのパラメータ `-upgrade-test-suite` *`flag`* を使用できます。常に最新のバージョンをダウンロードするには *`flag`* を「`y`」にし、既存のバージョンを使用するには「`n`」にします。

また、`list-supported-versions` コマンドを使用して、現在のバージョンの IDT でサポートされている FreeRTOS およびテストスイートのバージョンを一覧表示することもできます。

新しいテストでは、新しい IDT 構成設定が導入される可能性があります。その設定がオプションの場合は、IDT から通知され、テストの実行が続行されます。その設定が必要な場合は、IDT から通知され、実行が停止します。その設定を構成したら、テストの実行を続行できます。

エラーのトラブルシューティング

各テストスイートの実行には一意の実行 ID があります。この ID を使用して、`results` ディレクトリに `results/`*`execution-id`* という名前のフォルダを作成します。テストグループ別のログは `results/`*`execution-id`*`/logs` ディレクトリにあります。IDT for FreeRTOS コンソールの出力を使用して、失敗したテストケースの実行 ID、テストケース ID、およびテストグループ ID を検索し、そのテストケースの `results/`*`execution-id`*`/logs/`*`test_group_id`*`__`*`test_case_id`*`.log` という名前のログファイルを開きます。このファイルの情報には以下が含まれます。

- すべてのビルドおよびフラッシュコマンド出力。
- テスト実行出力。
- 詳細な IDT for FreeRTOS コンソール出力。

トラブルシューティングに次のワークフローをお勧めします。

1. 「*`user/role`* is not authorized to access this resource (user/role にこのリソースにアクセスする権限がありません)」というエラーが表示された場合、AWS アカウントの作成と設定 で指定したようにアクセス許可を設定していることを確認します。
2. コンソール出力を読み取り、実行 UUID や現在実行中のタスクなどの情報を確認します。
3. 各テストのエラーステートメントについて `FRQ_Report.xml` ファイルを確認します。このディレクトリには、各テストグループの実行ログが含まれています。
4. `/results/`*`execution-id`*`/logs` にあるログファイルを確認します。
5. 以下の問題領域のいずれかを調べてください。
   - `/configs/` フォルダの JSON 設定ファイルなどのデバイス設定。
   - デバイスインターフェイス。ログを確認して、どのインターフェイスが失敗しているかを判断します。
   - デバイスツール。デバイスをビルドおよびフラッシュするためのツールチェーンがインストールされ、正しく設定されていることを確認します。
   - FRQ 1.x.x では、FreeRTOS ソースコードのクローンされたクリーンなバージョンが使用可能であることを確認します。FreeRTOS リリースは FreeRTOS バージョンに従ってタグ付けされます。そのコードの特定のバージョンのクローンを作成するには、次のコマンドを使用します。

```
git clone --branch version-number https://github.com/aws/amazon-freertos.git
cd amazon-freertos
git submodule update --checkout --init --recursive
```

デバイス設定のトラブルシューティング

IDT for FreeRTOS を使用するときは、バイナリを実行する前に正しい設定ファイルを所定の場所に配置する必要があります。解析エラーや設定エラーが発生する場合は、まず環境に適した設定テンプレートを見つけて使用してください。これらのテンプレートは、*IDT_ROOT*/configs ディレクトリにあります。

それでも問題が解決されない場合は、次のデバッグプロセスを参照してください。

どこを見ればよいか

まず、コンソール出力を読み取って、このドキュメントで execution-id として参照される実行 UUID などの情報を見つけます。

次に、/results/*execution-id* ディレクトリにある FRQ_Report.xml ファイルを確認します。このファイルには、実行されたすべてのテストケースと、各障害のエラースニペットがあります。すべての実行ログを取得するには、各テストケースの /results/*execution-id*/logs/*test_group_id*__*test_case_id*.log ファイルを探します。

IDT エラーコード

IDT for FreeRTOS によって生成されるエラーコードを次の表に示します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
201	InvalidInputError	device.json 、config.json 、または userdata.json のフィールドがないか、正しくない形式です。	必須フィールドが欠落していないこと、およびリストされたファイルで必須の形式であることを確認します。詳細については、マイクロコン

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			トローラーボードの最初のテスト を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
202	ValidationError	`device.json` 、`config.json` 、または `userdata.json` のフィールドに無効な値が含まれています。	レポートのエラーコードの右側にあるエラーメッセージを確認します。 • 無効な AWS リージョン - `config.json` ファイルで有効な AWS リージョンを指定します。AWS リージョンの詳細については、「リージョンとエンドポイント」を参照してください。 • 無効な AWS 認証情報 - テストマシンに有効な AWS 認証情報を設定します (環境変数または認証情報ファイルを使用)。認証フィールドが正しく設定されていることを確認します。詳細については、AWS アカウントの作成と設定 を参照してください。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
203	CopySourceCodeError	指定されたディレクトリに FreeRTOS ソースコードをコピーできません。	以下の項目について確認してください。 • `userdata.json` ファイルで有効な `sourcePath` が指定されていることを確認してください。 • FreeRTOS ソースコードディレクトリの下にある `build` フォルダを削除します (存在する場合)。詳細については、「ビルド、フラッシュ、テスト設定を設定する」を参照してください。 • Windows では、ファイルパス名の文字数制限があります。ファイルパス名が長いと、エラーが発生します。

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
204	BuildSourceError	FreeRTOS ソースコードをコンパイルできません。	以下の項目について確認してください。 • `userdata.json` ファイルの下にある `buildTool` 情報が正しいことを確認します。 • `cmake` をビルドツールとして使用している場合は、`buildTool` コマンドで `{{enableTests}}` が指定されていることを確認します。詳細については、ビルド、フラッシュ、テスト設定を設定する を参照してください。 • スペースを含むシステム上のファイルパス (`C:\Users\My Name\Desktop\` など) に IDT for FreeRTOS を抽出した場合は、パスが適切に解析されるように、ビルドコマンド内に追加の引用符が必要

エラーコード	エラーコード名	考えられる根本原因	トラブルシューティング
			になる場合があります。フラッシュコマンドにも同じことが必要な場合があります。
205	FlashOrRunTestError	IDT FreeRTOS が DUT で FreeRTOS をフラッシュまたは実行できません。	`userdata.json` ファイルの下にある `flashTool` 情報が正しいことを確認します。詳細については、ビルド、フラッシュ、テスト設定を設定する を参照してください。
2.0.6	StartEchoServerError	IDT FreeRTOS が、WiFi またはセキュアソケットテストでエコーサーバーを起動できません。	`userdata.json` ファイルの `echoServerConfiguration` で設定されたポートが使用中でないこと、ファイアウォールまたはネットワーク設定によってブロックされていないことを確認します。

設定ファイルの解析エラーをデバッグする

場合によっては、JSON 設定のタイプミスが解析エラーにつながることがあります。ほとんどの場合、JSON ファイルで括弧やカンマ、引用符を忘れたことが原因です。IDT for FreeRTOS は、JSON 検証を行い、デバッグ情報を出力します。エラーが発生した行、構文エラーの行番号と列番号が出力されます。この情報だけでエラーの修正が可能なはずですが、それでもエラーを特定でき

ない場合は、IDE、テキストエディタ (Atom、Sublime など)、またはオンラインツール (JSONLint など) を使って手動で検証を行います。

テスト結果解析エラーのデバッグ

FreeRTOS-Libraries-Integration-Tests からのテストグループ (FullTransportInterfaceTLS, FullPKCS11_Core、FullPKCS11_Onboard_ECC、FullPKCS11_Onboard_RSA、FullPKCS11_PreProvisione など) を実行している間、IDT for FreeRTOS はシリアル接続を使用してテストデバイスからのテスト結果を解析します。場合によっては、デバイス上の余分なシリアル出力がテスト結果の解析を妨げることがあります。

上記のケースでは、関係のないデバイス出力から派生した文字列など、テストケースの失敗に関する奇妙な理由が出力されます。IDT for FreeRTOS テストケースのログファイル (IDT for FreeRTOS がテスト中に受信したすべてのシリアル出力を含む) には、次の内容が表示される場合があります。

```
<unrelated device output>
TEST(Full_PKCS11_Capabilities, PKCS11_Capabilities)<unrelated device output>
<unrelated device output>
 PASS
```

上記の例では、関連のないデバイス出力により、IDT for FreeRTOS はテスト結果である PASS を検出できません。

次の点を確認して、最適なテストを行ってください。

- デバイスで使用されているロギングマクロがスレッドセーフであることを確認してください。詳細については、「Implementing the library logging macros」を参照してください。
- テスト中は、シリアル接続への出力が最小限になるようにしてください。ロギングマクロが適切にスレッドセーフであっても、テスト結果はテスト中に別々の呼び出しで出力されるため、他のデバイス出力は問題になる可能性があります。

IDT for FreeRTOS のテストケースログでは、次のような中断のないテスト結果出力が表示されることが理想的です。

```
---------STARTING TESTS---------
TEST(Full_OTA_PAL, otaPal_CloseFile_ValidSignature) PASS
TEST(Full_OTA_PAL, otaPal_CloseFile_InvalidSignatureBlockWritten) PASS
```

```
-----------------------
2 Tests 0 Failures 0 Ignored
```

整合性チェックの失敗のデバッグ

FRQ 1.x.x バージョンの FreeRTOS を使用している場合は、以下の整合性チェックが適用されます。

FreeRTOSIntegrity テストグループを実行して失敗した場合は、まず、*freertos* ディレクトリのファイルのいずれかを変更していないか確認します。変更しておらず、問題が解消しない場合は、正しいブランチを使用していることを確認してください。IDT の `list-supported-products` コマンドを実行すると、使用する必要のある *freertos* リポジトリのタグ付きブランチを検索できます。

`freertos` リポジトリの正しいタグ付きブランチのクローンを作成しても問題が解消しない場合は、`submodule update` コマンドも実行したかどうかを確認します。`freertos` リポジトリのクローンワークフローは次のとおりです。

```
git clone --branch version-number https://github.com/aws/amazon-freertos.git
cd amazon-freertos
git submodule update --checkout —init —recursive
```

整合性チェッカーが検索するファイルのリストは、*freertos* ディレクトリの `checksums.json` ファイルにあります。FreeRTOS の移植でファイルやフォルダの構造が変更されていないか確認するには、`checksums.json` ファイルの「`exhaustive`」と「`minimal`」セクションに記載されているファイルが変更されていないことを確認します。SDK を設定して実行するには、「`minimal`」セクションのファイルが変更されていないことを確認します。

SDK で IDT を実行する場合で、*freertos* ディレクトリ内のファイルを変更した場合、`userdata` ファイルの SDK が正しく設定されていることを確認します。それ以外の場合、整合性チェッカーは *freertos* ディレクトリ内のすべてのファイルを検証します。

FullWiFi テストグループの障害をデバッグする

FRQ 1.x.x を使用していて FullWiFi テストグループでエラーが発生し、「`AFQP_WiFiConnectMultipleAP`」テストで失敗する場合、両方のアクセスポイントが IDT を実行するホストコンピュータと同じサブネット内にない可能性があります。両方のアクセスポイントが IDT を実行するホストコンピュータと同じサブネット内にあることを確認してください。

「必須パラメータがありません」エラーのデバッグ

IDT for FreeRTOS には新機能が追加されているため、設定ファイルに変更が生じる可能性があります。古い設定ファイルを使用すると、設定が破損する可能性があります。このような場合は、`results/`*`execution-id`*`/logs` ディレクトリにある *`test_group_id`*`__`*`test_case_id`*`.log` ファイルに、すべての不足しているパラメータが明示的に一覧表示されます。IDT for FreeRTOS では、JSON 設定ファイルのスキーマを検証し、最新のサポートされているバージョンが使用されていることを確認します。

「テストを開始できませんでした」エラーのデバッグ

テストの開始中の障害を示唆するエラーが生じることがあります。原因はいくつか考えられるため、以下が正しいことを確認してください。

- 実行コマンドに含めたプール名が実際に存在することを確認します。この名前は、`device.json` ファイルから直接参照されます。
- プール内のデバイスの設定パラメータが正しいことを確認します。

「テスト結果の開始を見つけることができません」エラーをデバッグする

IDT がテスト対象のデバイスによって出力された結果を解析しようとすると、エラーが発生する場合があります。原因はいくつか考えられるため、以下の事項を確認して修正してください。

- テスト対象のデバイスがホストマシンに安定して接続していることを確認します。これらのエラーが発生するテストのログファイルで IDT が何を受信しているかを確認できます。
- FRQ 1.x.x を使用しており、テスト対象のデバイスが低速なネットワークまたはその他のインターフェイスを介して接続されている場合、または FreeRTOS テストグループの ログで「---------STARTING TESTS---------」フラグと他の FreeRTOS テストグループの出力が見つからない場合は、userdata 設定で `testStartDelayms` の値を増やしてみてください。詳細については、「ビルド、フラッシュ、テスト設定を設定する」を参照してください。

「テストの失敗: 予期された __ の結果はあるが、" エラーが表示される」のデバッグ

テスト中に、テストの失敗を示すエラーが表示される場合があります。テストで一定数の結果が期待されているにもかかわらず、テスト中にその結果を確認できません。一部の FreeRTOS テストが、IDT がデバイスからの出力を確認する前に実行されています。このエラーが表示された場合は、userdata 設定の `testStartDelayms` 値を増やしてみてください。詳細については、「ビルド、フラッシュ、テスト設定を設定する」を参照してください。

ConditionalTests の制約のため、"""" が選択されませんでした」エラーをデバッグする

これは、テストと互換性のないデバイスプールでテストを実行していることを意味します。これは、OTA E2E テストで発生することがあります。例えば、OTADataplaneMQTT テストグループを実行する際に、device.json 設定ファイルで OTA に No を選択したり、OTADataPlaneProtocol に HTTP を選択したりした場合などです。実行するテストグループは、device.json で選択した機能と一致している必要があります。

デバイス出力モニタリング中の IDT タイムアウトのデバッグ

IDT は、さまざまな理由でタイムアウトすることがあります。テストのデバイス出力モニタリングフェーズ中にタイムアウトが発生し、その結果を IDT のテストケースログ内で確認できる場合は、結果が IDT によって誤って解析されたことを意味します。その理由の 1 つとして、テスト結果の途中にインターリーブされたログメッセージが入っていることが考えられます。このような場合は、「FreeRTOS 移植ガイド」を参照して、UNITY ログの設定方法の詳細を確認してください。

デバイス出力モニタリング中にタイムアウトになるもう 1 つの理由として、TLS テストケースが 1 回失敗するとデバイスが再起動されることが考えられます。その後、デバイスはフラッシュされたイメージを実行し、これによって無限ループが発生します。これはログで確認できます。これが発生した場合は、テストに失敗した後にデバイスが再起動しないようにしてください。

「リソースへのアクセスが許可されていない」エラーをデバッグする

「*####/###*には、このリソースにアクセスする権限がありません」エラーがターミナル出力または /results/*execution-id*/logs の下の test_manager.log ファイルに表示される場合があります。この問題を解決するには、AWS IoTDeviceTesterForFreeRTOSFullAccess 管理ポリシーをテストユーザーにアタッチします。詳細については、「AWS アカウントの作成と設定」を参照してください。

ネットワークテストエラーのデバッグ

ネットワークベースのテストの場合、IDT は、ホストマシン上の予約されていないポートに結合するエコーサーバーを起動します。WiFi またはセキュアソケットテストでタイムアウトまたは接続不可のために エラーが発生した場合は、1024 ～ 49151 の範囲の設定済みポートへのトラフィックを許可するようにネットワークが設定されていることを確認します。

セキュアソケットテストでは、デフォルトでポート 33333 および 33334 が使用されます。WiFi テストでは、デフォルトでポート 33335 が使用されます。これら 3 つのポートが使用中であるか、ファイアウォールまたはネットワークによってブロックされている場合は、userdata.json でテスト用に異なるポートを使用することを選択できます。詳細については、ビルド、フラッシュ、テスト設定を

設定する を参照してください。以下のコマンドを使用して、特定のポートが使用中であるかどうかを確認できます。

- Windows: `netsh advfirewall firewall show rule name=all | grep port`
- Linux: `sudo netstat -pan | grep port`
- macOS: `netstat -nat | grep port`

同じバージョンのペイロードによる OTA 更新の失敗

OTA が実行された後にデバイス上に同じバージョンが存在するために OTA テストケースが失敗する場合、ビルドシステム (cmake など) が IDT の FreeRTOS ソースコード変更を認識せず、更新されたバイナリを構築していないことが原因である可能性があります。これにより、現在デバイス上にあるバイナリと同じバイナリで OTA が実行され、テストは失敗します。OTA 更新失敗のトラブルシューティングを行うには、まずサポートされている最新バージョンのビルドシステムを使用しているかを確認します。

`PresignedUrlExpired` テストケースの OTA テスト失敗

このテストの前提条件の 1 つは、OTA の更新時間が 60 秒を超えていることです。そうしないと、テストは失敗します。この問題が発生した場合、次のエラーメッセージがログに検出されます。「テストには 60 秒 (url の有効期限) 未満かかります。お気軽にご連絡ください。」

デバイスインターフェイスとポートエラーのデバッグ

このセクションでは、IDT がデバイスとの接続に使用するデバイスインターフェイスについて説明します。

サポートされているプラットフォーム

IDT は、Linux、macOS、Windows をサポートしています。これら 3 つのプラットフォームは、アタッチされるシリアルデバイスに対して異なる命名スキームを設けています。

- Linux: `/dev/tty*`
- macOS: `/dev/tty.*` または `/dev/cu.*`
- Windows: COM*

デバイスポートを確認するには:

- Linux/macOS の場合は、ターミナルを開き、`ls /dev/tty*` を実行します。

- macOS の場合は、ターミナルを開き、`ls /dev/tty.*` または `ls /dev/cu.*` を実行します。
- Windows の場合は、デバイスマネージャを開き、シリアルデバイスグループを展開します。

どのデバイスがポートに接続されているかを確認するには:

- Linux の場合、udev パッケージがインストールされていることを確認してから、`udevadm info --name=`*`PORT`* を実行します。このユーティリティにより、正しいポートを使用していることを確認するのに役立つではデバイスドライバー情報が出力されます。
- macOS の場合、Launchpad を開いて **`System Information`** を検索します。
- Windows の場合は、デバイスマネージャを開き、シリアルデバイスグループを展開します。

デバイスインターフェイス

組み込みデバイスはそれぞれに異なり、シリアルポートを 1 つ持つものもあれば、複数持つものもあります。一般的に、マシンへの接続時に次の 2 つのポートがデバイスに割り当てられます。

- デバイスをフラッシュするためのデータポート。
- 出力を読み取る読み取りポート。

  `device.json` ファイルで適切な読み取りポートを設定する必要があります。そのように設定しない場合は、デバイスからの出力の読み取りが失敗する可能性があります。

  複数のポートがある場合、必ず `device.json` ファイルにあるデバイスの読み取りポートを使用してください。たとえば、Espressif WRover デバイスを接続し、デバイスに `/dev/ttyUSB0` と `/dev/ttyUSB1` の 2 つのポートが割り当てられている場合、`device.json` ファイルでは `/dev/ttyUSB1` を使用します。

Windows の場合は、同じロジックに従います。

デバイスデータの読み取り

IDT for FreeRTOS は、個々のデバイスのビルドおよびフラッシュツールを使用してポート設定を指定します。デバイスをテストしていて、出力が取得できない場合は、次のようなデフォルト設定を試してみてください。

- ボーレート: 115200
- データビット: 8

- パリティ: なし
- ストップビット: 1
- フロー制御: なし

これらの設定は、IDT for FreeRTOS によって処理されます。それらを設定する必要はありません。ただし、同じ方法を使用して手動でデバイス出力を読み取ることができます。Linux または macOS では、`screen` コマンドを使用して行います。Windows では、TeraTerm などのプログラムを使用します。

```
Screen: screen /dev/cu.usbserial 115200
```

```
TeraTerm: Use the above-provided settings to set the fields explicitly in
the GUI.
```

開発ツールチェーンの問題

このセクションでは、ツールチェーンで生じる可能性のある問題を取り上げます。

Ubuntu での Code Composer Studio

それより新しいバージョンの Ubuntu (17.10 と 18.04) だと、`glibc` パッケージのバージョンに Code Composer Studio 7.x バージョンとの互換性がありません。Code Composer Studio version 8.2 以降をインストールすることをお勧めします。

互換性がない場合、次のような症状が現れます。

- FreeRTOS がビルドやデバイスへのフラッシュに失敗する。
- Code Composer Studio インストーラがフリーズする。
- ビルドまたはフラッシュ中に、コンソールにログ出力が表示されない。
- ヘッドレスとして呼び出したビルドコマンドが GUI モードで起動しようとする。

ロギング

IDT for FreeRTOS のログは 1 箇所に配置されます。ルート IDT ディレクトリから、`results/`*`execution-id`*`/` の以下のファイルにアクセスできます。

- `FRQ_Report.xml`
- `awsiotdevicetester_report.xml`

- logs/*test_group_id*__*test_case_id*.log

FRQ_Report.xml と logs/*test_group_id*__*test_case_id*.log は、調査すべき最も重要なログです。FRQ_Report.xml には、特定のエラーメッセージで失敗したテストケースに関する情報が含まれています。次に、logs/*test_group_id*__*test_case_id*.log を使用して問題の詳細を確認し、状況を把握します。

コンソールエラー

AWS IoT Device Tester が実行されると、障害は短いメッセージとともにコンソールに報告されます。results/*execution-id*/logs/*test_group_id*__*test_case_id*.log でエラーの詳細を確認します。

ログエラー

各テストスイートの実行には一意の実行 ID があり、これを使用して results/*execution-id* という名前のフォルダを作成します。テストケース別のログは results/*execution-id*/logs ディレクトリにあります。IDT for FreeRTOS コンソールの出力を使用して、失敗したテストケースの実行 ID、テストケース ID、およびテストグループ ID を検索します。次に、この情報を使用して、 という名前のテストケースのログファイルを見つけて開きますresults/*execution-id*/logs/*test_group_id*__*test_case_id*.log。このファイルの情報には、完全なビルドおよびフラッシュコマンド出力、テスト実行出力、およびより詳細な AWS IoT Device Tester コンソール出力が含まれます。

S3 バケットの問題

IDT の実行中に CTRL+C を押すと、クリーンアッププロセスが開始されます。このクリーンアップには、IDT テストの一部として作成された Amazon S3 リソースの削除が含まれます。クリーンアップを完了できない場合、Amazon S3 バケットが過剰に作成される問題が発生することがあります。つまり、次回 IDT を実行したときにテストが失敗し始めます。

IDT を停止するために CTRL+C を押す場合、この問題を回避するためにクリーンアッププロセスを完了させる必要があります。アカウントから手動で作成された Amazon S3 バケットを削除することもできます。

タイムアウトエラーのトラブルシューティング

テストスイートの実行中にタイムアウトエラーが発生した場合は、タイムアウト乗数を指定してタイムアウトを増やします。この乗数は、デフォルトのタイムアウト値に適用されます。このフラグに設

定された値はすべて、1.0 以上である必要があります。タイムアウトの乗数を使用するには、テストスイートの実行時に `--timeout-multiplier` フラグを使用します。

Example

IDT v3.0.0 and later

```
./devicetester_linux run-suite --suite-id FRQ_1.0.1 --pool-id DevicePool1 --timeout-multiplier 2.5
```

IDT v1.7.0 and earlier

```
./devicetester_linux run-suite --suite-id FRQ_1 --pool-id DevicePool1 --timeout-multiplier 2.5
```

セルラー機能と AWS 料金

`device.JSON` ファイル`Yes`で `Cellular`機能が に設定されている場合、FullSecureSockets はテストの実行に t.micro EC2 インスタンスを使用します。これにより、 AWS アカウントに追加料金が発生する可能性があります。詳細については「アマゾン EC2 料金」を参照してください。

認定レポート生成ポリシー

資格レポートは、過去 2 年間にリリースされた FreeRTOS バージョンをサポートする AWS IoT Device Tester (IDT) バージョンによってのみ生成されます。サポートポリシーについてご質問がある場合は、 AWS サポート までお問い合わせください。

の AWS 管理ポリシーを理解する AWS IoT Device Tester

AWS 管理ポリシーは、 によって作成および管理されるスタンドアロンポリシーです AWS。 AWS 管理ポリシーは、多くの一般的なユースケースにアクセス許可を付与するように設計されているため、ユーザー、グループ、ロールにアクセス許可の割り当てを開始できます。

AWS 管理ポリシーは、すべての AWS お客様が使用できるため、特定のユースケースに対して最小特権のアクセス許可を付与しない場合があることに注意してください。ユースケースに固有のカスタマー管理ポリシーを定義して、アクセス許可を絞り込むことをお勧めします。

AWS 管理ポリシーで定義されているアクセス許可は変更できません。が AWS 管理ポリシーで定義されたアクセス許可 AWS を更新すると、ポリシーがアタッチされているすべてのプリンシパル ID

(ユーザー、グループ、ロール) に影響します。 AWS は、新しい AWS のサービス が起動されたとき、または既存のサービスで新しい API オペレーションが利用可能になったときに、 AWS 管理ポリシーを更新する可能性が最も高くなります。

詳細については「IAM ユーザーガイド」の「AWS マネージドポリシー」を参照してください。

トピック

- AWS マネージドポリシー： AWS IoTDeviceTesterForFreeRTOSFullAccess
- AWS 管理ポリシーの更新

AWS マネージドポリシー： AWS IoTDeviceTesterForFreeRTOSFullAccess

`AWSIoTDeviceTesterForFreeRTOSFullAccess` 管理ポリシーには、バージョンチェック、自動更新機能、メトリクスの収集に関する以下の AWS IoT Device Tester アクセス許可が含まれています。

アクセス許可の詳細

このポリシーには、以下のアクセス許可が含まれています。

- `iot-device-tester:SupportedVersion`

  サポートされている製品、テストスイート、IDT バージョンのリストを取得する AWS IoT Device Tester 許可を付与。
- `iot-device-tester:LatestIdt`

  ダウンロード可能な最新の IDT バージョンを取得する AWS IoT Device Tester 許可を付与。
- `iot-device-tester:CheckVersion`

  IDT、テストスイート、製品のバージョンの互換性をチェックする AWS IoT Device Tester 許可を付与。
- `iot-device-tester:DownloadTestSuite`

  テストスイートの更新をダウンロードする AWS IoT Device Tester 許可を付与。
- `iot-device-tester:SendMetrics`

  AWS IoT Device Tester 内部使用状況に関するメトリクスを収集する AWS 許可を付与。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::*:role/idt-*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": "iot.amazonaws.com"
                }
            }
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "iot:DeleteThing",
                "iot:AttachThingPrincipal",
                "iot:DeleteCertificate",
                "iot:GetRegistrationCode",
                "iot:CreatePolicy",
                "iot:UpdateCACertificate",
                "s3:ListBucket",
                "iot:DescribeEndpoint",
                "iot:CreateOTAUpdate",
                "iot:CreateStream",
                "signer:ListSigningJobs",
                "acm:ListCertificates",
                "iot:CreateKeysAndCertificate",
                "iot:UpdateCertificate",
                "iot:CreateCertificateFromCsr",
                "iot:DetachThingPrincipal",
                "iot:RegisterCACertificate",
                "iot:CreateThing",
                "iam:ListRoles",
                "iot:RegisterCertificate",
                "iot:DeleteCACertificate",
                "signer:PutSigningProfile",
                "s3:ListAllMyBuckets",
                "signer:ListSigningPlatforms",
```

```
                "iot-device-tester:SendMetrics",
                "iot-device-tester:SupportedVersion",
                "iot-device-tester:LatestIdt",
                "iot-device-tester:CheckVersion",
                "iot-device-tester:DownloadTestSuite"
            ],
            "Resource": "*"
        },
        {
            "Sid": "VisualEditor2",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "signer:StartSigningJob",
                "acm:GetCertificate",
                "signer:DescribeSigningJob",
                "s3:CreateBucket",
                "execute-api:Invoke",
                "s3:DeleteBucket",
                "s3:PutBucketVersioning",
                "signer:CancelSigningProfile"
            ],
            "Resource": [
                "arn:aws:execute-api:us-east-1:098862408343:9xpmnvs5h4/prod/POST/
metrics",
                "arn:aws:signer:*:*:/signing-profiles/*",
                "arn:aws:signer:*:*:/signing-jobs/*",
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:acm:*:*:certificate/*",
                "arn:aws:s3:::idt-*",
                "arn:aws:s3:::afr-ota*"
            ]
        },
        {
            "Sid": "VisualEditor3",
            "Effect": "Allow",
            "Action": [
                "iot:DeleteStream",
                "iot:DeleteCertificate",
                "iot:AttachPolicy",
                "iot:DetachPolicy",
                "iot:DeletePolicy",
                "s3:ListBucketVersions",
                "iot:UpdateCertificate",
```

```
                "iot:GetOTAUpdate",
                "iot:DeleteOTAUpdate",
                "iot:DescribeJobExecution"
            ],
            "Resource": [
                "arn:aws:s3:::afr-ota*",
                "arn:aws:iot:*:*:thinggroup/idt*",
                "arn:aws:iam::*:role/idt-*"
            ]
        },
        {
            "Sid": "VisualEditor4",
            "Effect": "Allow",
            "Action": [
                "iot:DeleteCertificate",
                "iot:AttachPolicy",
                "iot:DetachPolicy",
                "s3:DeleteObjectVersion",
                "iot:DeleteOTAUpdate",
                "s3:PutObject",
                "s3:GetObject",
                "iot:DeleteStream",
                "iot:DeletePolicy",
                "s3:DeleteObject",
                "iot:UpdateCertificate",
                "iot:GetOTAUpdate",
                "s3:GetObjectVersion",
                "iot:DescribeJobExecution"
            ],
            "Resource": [
                "arn:aws:s3:::afr-ota*/*",
                "arn:aws:s3:::idt-*/*",
                "arn:aws:iot:*:*:policy/idt*",
                "arn:aws:iam::*:role/idt-*",
                "arn:aws:iot:*:*:otaupdate/idt*",
                "arn:aws:iot:*:*:thing/idt*",
                "arn:aws:iot:*:*:cert/*",
                "arn:aws:iot:*:*:job/*",
                "arn:aws:iot:*:*:stream/*"
            ]
        },
        {
            "Sid": "VisualEditor5",
            "Effect": "Allow",
```

```
            "Action": [
                "s3:PutObject",
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::afr-ota*/*",
                "arn:aws:s3:::idt-*/*"
            ]
        },
        {
            "Sid": "VisualEditor6",
            "Effect": "Allow",
            "Action": [
                "iot:CancelJobExecution"
            ],
            "Resource": [
                "arn:aws:iot:*:*:job/*",
                "arn:aws:iot:*:*:thing/idt*"
            ]
        },
        {
            "Sid": "VisualEditor7",
            "Effect": "Allow",
            "Action": [
                "ec2:TerminateInstances"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:instance/*"
            ],
            "Condition": {
                "StringEquals": {
                    "ec2:ResourceTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
            "Sid": "VisualEditor8",
            "Effect": "Allow",
            "Action": [
                "ec2:AuthorizeSecurityGroupIngress",
                "ec2:DeleteSecurityGroup"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:security-group/*"
```

```
            ],
            "Condition": {
                "StringEquals": {
                    "ec2:ResourceTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
            "Sid": "VisualEditor9",
            "Effect": "Allow",
            "Action": [
                "ec2:RunInstances"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:instance/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
            "Sid": "VisualEditor10",
            "Effect": "Allow",
            "Action": [
                "ec2:RunInstances"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:image/*",
                "arn:aws:ec2:*:*:security-group/*",
                "arn:aws:ec2:*:*:volume/*",
                "arn:aws:ec2:*:*:key-pair/*",
                "arn:aws:ec2:*:*:placement-group/*",
                "arn:aws:ec2:*:*:snapshot/*",
                "arn:aws:ec2:*:*:network-interface/*",
                "arn:aws:ec2:*:*:subnet/*"
            ]
        },
        {
            "Sid": "VisualEditor11",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateSecurityGroup"
```

```
            ],
            "Resource": [
                "arn:aws:ec2:*:*:security-group/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/Owner": "IoTDeviceTester"
                }
            }
        },
        {
            "Sid": "VisualEditor12",
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeInstances",
                "ec2:DescribeSecurityGroups",
                "ssm:DescribeParameters",
                "ssm:GetParameters"
            ],
            "Resource": "*"
        },
        {
            "Sid": "VisualEditor13",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateTags"
            ],
            "Resource": [
                "arn:aws:ec2:*:*:security-group/*",
                "arn:aws:ec2:*:*:instance/*"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "aws:TagKeys": [
                        "Owner"
                    ]
                },
                "StringEquals": {
                    "ec2:CreateAction": [
                        "RunInstances",
                        "CreateSecurityGroup"
                    ]
                }
            }
```

```
        }
    ]
}
```

AWS 管理ポリシーの更新

の AWS マネージドポリシーの更新に関する詳細は、このサービスがこれらの変更の追跡を開始した時点 AWS IoT Device Tester から確認できます。

バージョン	変更	説明	日付
7 (最新)	`ec2:CreateTags` 条件を再構築しました。	`ForAnyValues` の使用法を削除しました。	6/14/2023
6	ポリシーから `freertos:ListHardwarePlatforms` が削除されました。	このアクションは 2023 年 3 月 1 日に廃止されたため、権限を削除します。	6/2/2023
5	EC2 を使用してエコーサーバーテストを実行する権限を追加しました。	これは、顧客の AWS アカウントで EC2 インスタンスを起動および停止するためのものです。	12/15/2020
4	`iot:CancelJobExecution` が追加されました。	この権限は OTA ジョブをキャンセルします。	7/17/2020
3	以下の権限を追加しました。 • `iot-device-tester:DownloadTestSuite` ,	• `iot-device-tester:DownloadTestSuite` — テストスイートの更新をダウンロードする AWS IoT Device	3/23/2020

バージョン	変更	説明	日付
	• iot-device-tester:CheckVersion , • iot-device-tester:LatestIdt , • iot-device-tester:SupportedVersion .	Tester アクセス許可を付与します。 • iot-device-tester:CheckVersion — IDT、テストスイート、製品のバージョンの互換性をチェックする AWS IoT Device Tester 許可を付与 • iot-device-tester:LatestIdt — ダウンロード可能な最新の IDT バージョンを取得する AWS IoT Device Tester アクセス許可を付与します。 • iot-device-tester:SupportedVersion — サポートされている製品、テストスイート、IDT バージョンのリストを取得する AWS IoT Device Tester 許可を付与。	

バージョン	変更	説明	日付
2	`iot-device-tester: SendMetrics` 権限を追加しました。	AWS IoT Device Tester 内部使用状況に関するメトリクスを収集する AWS 許可を付与。	2/18/2020
1	当初のバージョン		2/12/2020

のサポートポリシーを理解する AWS IoT Device Tester

> **Important**
>
> 2022 年 10 月現在、 AWS IoT Device Tester for AWS IoT FreeRTOS Qualification (FRQ) 1.0 は署名付き認定レポートを生成しません。IDT FRQ 1.0 バージョンを使用してAWS 、デバイス認定プログラムを通じて、新しい AWS IoT FreeRTOS デバイスを AWS Partner Device Catalog にリストする資格を得ることはできません。IDT FRQ 1.0 を使用して FreeRTOS デバイスを認定することはできませんが、引き続き FRQ 1.0 を使用して FreeRTOS デバイスをテストすることはできます。FreeRTOS デバイスを認定し、AWS Partner Device Catalog にリストする際には、IDT FRQ 2.0 を使用することが推奨されます。

AWS IoT Device Tester for FreeRTOS は、デバイスへの FreeRTOS ポートを検証するためのテスト自動化ツールです。さらに、FreeRTOS デバイスを認定し、AWS Partner Device Catalog にリストすることもできます。 AWS IoT Device Tester for FreeRTOS は、GitHub の FreeRTOS/FreeRTOS-LTS で利用可能な FreeRTOS 長期サポート (LTS) ライブラリの検証と認定、および FreeRTOS/FreeRTOS で利用可能な FreeRTOS メインラインをサポートしています。FreeRTOS と AWS IoT Device Tester for FreeRTOS の両方の最新バージョンを使用して、デバイスを検証および認定することをお勧めします。

FreeRTOS-LTS については、IDT は FreeRTOS 202210 LTS バージョンの検証と認定をサポートしています。FreeRTOS LTS のリリースとメンテナンスのタイムラインの詳細については、こちらを参照してください。これらの LTS リリースのサポート期間が終了しても検証を継続することはできますが、IDT はデバイスの認定を依頼するために必要なレポートを生成しません。

FreeRTOS/FreeRTOS で入手可能なメインラインの FreeRTOS については、過去 6 か月にリリースされたすべてのバージョン、またはリリースの間隔が 6 か月以上空いている場合は過去 2 つのバージョンの FreeRTOS の検証と認定をサポートしています。現在サポートされているバージョンについては、こちらを参照してください。サポート対象外のバージョンの FreeRTOS の場合、検証を継続することはできますが、IDT はデバイスの認定を依頼するために必要なレポートを生成しません。

サポートされている IDT および FreeRTOS のバージョンの最新情報については、「のサポートされているバージョン AWS IoT Device Tester」を参照してください。サポートされているバージョンのを対応するバージョンの FreeRTOS AWS IoT Device Tester とともに使用して、デバイスをテストまたは認定できます。引き続き IDT for FreeRTOS のサポートされていないバージョン を使用する場合は、これらのバージョンのバグ修正や更新プログラムは受け取れません。

サポートポリシーについてご質問がある場合は、AWS カスタマーサポートまでお問い合わせください。

のセキュリティ AWS

のクラウドセキュリティが最優先事項 AWS です。お客様は AWS 、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャを活用できます。

セキュリティは、 AWS お客様とお客様の間の責任共有です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

- クラウドのセキュリティ – AWS クラウドで AWS サービスを実行するインフラストラクチャを保護する AWS 責任があります。 AWS また、 では、安全に使用できるサービスも提供しています。セキュリティの有効性は、AWS コンプライアンスプログラムの一環として、サードパーティーの審査機関によって定期的にテストおよび検証されています。 AWS サービスに適用されるコンプライアンスプログラムの詳細については、AWS「コンプライアンスプログラムによる対象範囲内のサービス」を参照してください。
- クラウドのセキュリティ – お客様の責任は、使用する AWS サービスによって決まります。また、お客様は、お客様のデータの機密性、組織の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

このドキュメントは、 を使用する際の責任共有モデルの適用方法を理解するのに役立ちます AWS。以下のトピックでは、セキュリティおよびコンプライアンスの目的を達成する AWS ように を設定する方法を示します。また、 AWS リソースのモニタリングと保護に役立つ AWS のサービスの使用方法についても説明します。

AWS IoT セキュリティの詳細については、「Security and Identity for AWS IoT」を参照してください。

トピック

- FreeRTOS の Identity and Access Management
- コンプライアンス検証
- の耐障害性 AWS
- FreeRTOS でのインフラストラクチャセキュリティ

FreeRTOS の Identity and Access Management

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービス するのに役立つ です。IAM 管理者は、FreeRTOS リソースを使用するための認証を受ける (サインインできる) ユーザーおよび認可を受ける (アクセス許可を持つ) ユーザーを制御します。IAM は、追加料金なしで使用できる AWS のサービス です。

トピック

- 対象者
- アイデンティティを使用した認証
- ポリシーを使用したアクセスの管理
- FreeRTOS と IAM の連携の仕組み
- FreeRTOS のアイデンティティベースのポリシーの例
- FreeRTOS ID とアクセスのトラブルシューティング

対象者

AWS Identity and Access Management (IAM) の使用方法は、FreeRTOS で行う作業によって異なります。

サービスユーザー – ジョブを実行するために FreeRTOS サービスを使用する場合は、管理者から必要な認証情報とアクセス許可が付与されます。作業を実行するためにさらに多くの FreeRTOS の機能を使用する場合は、追加のアクセス許可が必要になることがあります。アクセスの管理方法を理解すると、管理者に適切なアクセス許可をリクエストするのに役に立ちます。FreeRTOS の機能にアクセスできない場合は、「FreeRTOS ID とアクセスのトラブルシューティング」を参照してください。

サービス管理者 – 社内の FreeRTOS リソースの担当者であれば、通常、FreeRTOS へのフルアクセスがあるはずです。サービスユーザーがアクセスすべき FreeRTOS 機能やリソースを決定するのは、管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を点検して、IAM の基本概念を理解してください。FreeRTOS での IAM の使用方法の詳細については、「FreeRTOS と IAM の連携の仕組み」を参照してください。

IAM 管理者 – IAM 管理者には、FreeRTOS へのアクセスを管理するポリシーの作成方法の詳細を理解することが推奨されます。IAM で使用できる FreeRTOS のアイデンティティベースのポリシーの例を確認するには、「FreeRTOS のアイデンティティベースのポリシーの例」を参照してください。

アイデンティティを使用した認証

認証とは、ID 認証情報 AWS を使用して にサインインする方法です。として、IAM ユーザーとして AWS アカウントのルートユーザー、または IAM ロールを引き受けることによって、認証 (にサインイン AWS) される必要があります。

ID ソースを介して提供された認証情報を使用して、フェデレーティッド ID AWS として にサインインできます。 AWS IAM Identity Center （ IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook 認証情報は、フェデレーティッド ID の例です。フェデレーティッド ID としてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーション AWS を使用して にアクセスすると、間接的にロールを引き受けることになります。

ユーザーの種類に応じて、 AWS Management Console または AWS アクセスポータルにサインインできます。へのサインインの詳細については AWS、「 AWS サインイン ユーザーガイド」の「 へのサインイン AWS アカウント方法」を参照してください。

AWS プログラムで にアクセスする場合、 はソフトウェア開発キット (SDK) とコマンドラインインターフェイス (CLI) AWS を提供し、認証情報を使用してリクエストを暗号化して署名します。AWS ツールを使用しない場合は、自分でリクエストに署名する必要があります。リクエストに自分で署名する推奨方法の使用については、「IAM ユーザーガイド」の「API リクエストに対するAWS Signature Version 4」を参照してください。

使用する認証方法を問わず、追加セキュリティ情報の提供をリクエストされる場合もあります。たとえば、 では、アカウントのセキュリティを高めるために多要素認証 (MFA) を使用する AWS ことをお勧めします。詳細については、「AWS IAM Identity Center ユーザーガイド」の「多要素認証」および「IAM ユーザーガイド」の「IAM のAWS 多要素認証」を参照してください。

AWS アカウント ルートユーザー

を作成するときは AWS アカウント、アカウント内のすべての およびリソースへの AWS のサービス 完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。この ID は AWS アカウント ルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることでアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、「IAM ユーザーガイド」の「ルートユーザー認証情報が必要なタスク」を参照してください。

フェデレーティッドアイデンティティ

ベストプラクティスとして、管理者アクセスを必要とするユーザーを含む人間のユーザーに、ID プロバイダーとのフェデレーションを使用して一時的な認証情報 AWS のサービス を使用して にアクセスすることを要求します。

フェデレーティッド ID は、エンタープライズユーザーディレクトリ、ウェブ ID プロバイダー、AWS Directory Service、アイデンティティセンターディレクトリ、または ID ソースを介して提供された認証情報 AWS のサービス を使用して にアクセスするユーザーです。フェデレーティッド ID がにアクセスすると AWS アカウント、ロールを引き受け、ロールは一時的な認証情報を提供します。

アクセスを一元管理する場合は、 AWS IAM Identity Centerを使用することをお勧めします。IAM Identity Center でユーザーとグループを作成するか、独自の ID ソースのユーザーとグループのセットに接続して同期し、すべての AWS アカウント とアプリケーションで使用できます。IAM Identity Center の詳細については、「AWS IAM Identity Center ユーザーガイド」の「What is IAM Identity Center?」(IAM Identity Center とは) を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、単一のユーザーまたはアプリケーションに対して特定のアクセス許可 AWS アカウント を持つ 内の ID です。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時的な認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、「IAM ユーザーガイド」の「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する許可を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時認証情報が提供されます。詳細については、「IAM ユーザーガイド」の「IAM ユーザーに関するユースケース」を参照してください。

IAM ロール

IAM ロールは、特定のアクセス許可 AWS アカウント を持つ 内の ID です。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。で IAM ロールを一時的に引き受けるには AWS Management Console、ユーザーから IAM ロール (コンソール) に切り替えることができます。ロールを引き受けるには、 または AWS API オペレーションを AWS CLI 呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「ロールを引き受けるための各種方法」を参照してください。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッド ID に許可を割り当てるには、ロールを作成してそのロールの許可を定義します。フェデレーティッド ID が認証されると、その ID はロールに関連付けられ、ロールで定義されている許可が付与されます。フェデレーションのロールについては、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。IAM Identity Center を使用する場合は、許可セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。アクセス許可セットの詳細については、「AWS IAM Identity Center User Guide」の「Permission sets」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の では AWS のサービス、 (プロキシとしてロールを使用する代わりに) リソースに直接ポリシーをアタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。
- クロスサービスアクセス — 一部の は他の の機能 AWS のサービス を使用します AWS のサービス。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの許可、サービスロール、またはサービスリンクロールを使用してこれを行う場合があります。
  - 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用してアクションを実行すると AWS、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、 を呼び出すプリンシパルのアクセス許可を AWS のサービス、ダウンストリームサービス AWS のサービ

ス へのリクエストをリクエストする と組み合わせて使用します。FAS リクエストは、サービスが他の AWS のサービス またはリソースとのやり取りを完了する必要があるリクエストを受け取った場合にのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

- サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除することができます。詳細については、「IAM ユーザーガイド」の「AWS のサービスに許可を委任するロールを作成する」を参照してください。
- サービスにリンクされたロール – サービスにリンクされたロールは、 にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント 、サービスによって所有されます。IAM 管理者は、サービスリンクロールのアクセス許可を表示できますが、編集することはできません。

- Amazon EC2 で実行されているアプリケーション – IAM ロールを使用して、EC2 インスタンスで実行され、 AWS CLI または AWS API リクエストを行うアプリケーションの一時的な認証情報を管理できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。EC2 インスタンスに AWS ロールを割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスにアタッチされたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、「IAM ユーザーガイド」の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用して許可を付与する」を参照してください。

ポリシーを使用したアクセスの管理

でアクセスを制御する AWS には、ポリシーを作成し、ID AWS またはリソースにアタッチします。ポリシーは AWS 、アイデンティティまたはリソースに関連付けられているときにアクセス許可を定義する のオブジェクトです。 は、プリンシパル (ユーザー、ルートユーザー、またはロールセッション) がリクエストを行うときに、これらのポリシー AWS を評価します。ポリシーでの権限により、リクエストが許可されるか拒否されるかが決まります。ほとんどのポリシーは JSON ドキュメント AWS として に保存されます。JSON ポリシードキュメントの構造と内容の詳細については、「IAM ユーザーガイド」の「JSON ポリシー概要」を参照してください。

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き受けることができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの許可を定義します。例えば、`iam:GetRole` アクションを許可するポリシーがあるとします。そのポリシーを持つユーザーは、 AWS Management Console、、 AWS CLIまたは AWS API からロール情報を取得できます。

アイデンティティベースのポリシー

アイデンティティベースポリシーは、IAM ユーザーグループ、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティベースポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

アイデンティティベースのポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれています。管理ポリシーは、 内の複数のユーザー、グループ、ロールにアタッチできるスタンドアロンポリシーです AWS アカウント。管理ポリシーには、 AWS 管理ポリシーとカスタマー管理ポリシーが含まれます。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「管理ポリシーとインラインポリシーのいずれかを選択する」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または を含めることができます AWS のサービス。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは、IAM の AWS マネージドポリシーを使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Amazon S3、および Amazon VPC は AWS WAF、ACLs。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS は、一般的でない追加のポリシータイプをサポートします。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- アクセス許可の境界 - アクセス許可の境界は、アイデンティティベースポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる権限の上限を設定する高度な機能です。エンティティにアクセス許可の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとそのアクセス許可の境界の共通部分になります。`Principal` フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、アクセス許可の境界は制限されません。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。アクセス許可の境界の詳細については、「IAM ユーザーガイド」の「IAM エンティティのアクセス許可の境界」を参照してください。
- サービスコントロールポリシー (SCPs) – SCPs は、 の組織または組織単位 (OU) の最大アクセス許可を指定する JSON ポリシーです AWS Organizations。 AWS Organizations は、ビジネスが所有する複数の をグループ化して一元管理するためのサービス AWS アカウント です。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP は、各 を含むメンバーアカウントのエンティティのアクセス許可を制限します AWS アカウントのルートユーザー。Organizations と SCP の詳細については、「AWS Organizations ユーザーガイド」の「サービスコントロールポリシー (SCP)」を参照してください。
- リソースコントロールポリシー (RCP) – RCP は、所有する各リソースにアタッチされた IAM ポリシーを更新することなく、アカウント内のリソースに利用可能な最大数のアクセス許可を設定するために使用できる JSON ポリシーです。RCP は、メンバーアカウントのリソースのアクセス許可を制限し、組織に属しているかどうかにかかわらず AWS アカウントのルートユーザー、 を含む ID の有効なアクセス許可に影響を与える可能性があります。RCP をサポートする のリストを含む Organizations と RCP の詳細については、AWS Organizations RCPs「リソースコントロールポリシー (RCPs」を参照してください。 AWS のサービス

- セッションポリシー - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合もあります。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「セッションポリシー」を参照してください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。が複数のポリシータイプが関与する場合にリクエストを許可するかどうか AWS を決定する方法については、「IAM ユーザーガイド」の「ポリシー評価ロジック」を参照してください。

FreeRTOS と IAM の連携の仕組み

IAM を使用して FreeRTOS へのアクセスを管理する前に、FreeRTOS で使用できる IAM の機能について理解してください。

FreeRTOS で使用できる IAM の機能

IAM 機能	FreeRTOS サポート
アイデンティティベースポリシー	はい
リソースベースのポリシー	いいえ
ポリシーアクション	はい
ポリシーリソース	はい
ポリシー条件キー (サービス固有)	はい
ACL	いいえ
ABAC (ポリシー内のタグ)	部分的
一時的な認証情報	はい

IAM 機能	FreeRTOS サポート
プリンシパル権限	はい
サービスロール	はい
サービスリンクロール	いいえ

FreeRTOS およびその他の AWS のサービスがほとんどの IAM 機能と連携する方法の概要については、IAM ユーザーガイドのAWS 「IAM と連携する のサービス」を参照してください。

FreeRTOS のアイデンティティベースのポリシー

ID ベースのポリシーのサポート: あり

アイデンティティベースポリシーは、IAM ユーザーグループ、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。ID ベースのポリシーの作成方法については、「IAM ユーザーガイド」の「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」を参照してください。

IAM アイデンティティベースのポリシーでは、許可または拒否するアクションとリソース、およびアクションを許可または拒否する条件を指定できます。プリンシパルは、それが添付されているユーザーまたはロールに適用されるため、アイデンティティベースのポリシーでは指定できません。JSON ポリシーで使用できるすべての要素について学ぶには、「IAM ユーザーガイド」の「IAM JSON ポリシーの要素のリファレンス」を参照してください。

FreeRTOS のアイデンティティベースのポリシーの例

FreeRTOS のアイデンティティベースのポリシーの例を確認するには、「FreeRTOS のアイデンティティベースのポリシーの例」を参照してください。

FreeRTOS 内のリソースベースのポリシー

リソースベースのポリシーのサポート: なし

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの

場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、または を含めることができます AWS のサービス。

クロスアカウントアクセスを有効にするには、アカウント全体、または別のアカウントの IAM エンティティをリソースベースのポリシーのプリンシパルとして指定します。リソースベースのポリシーにクロスアカウントのプリンシパルを追加しても、信頼関係は半分しか確立されない点に注意してください。プリンシパルとリソースが異なる場合 AWS アカウント、信頼されたアカウントの IAM 管理者は、リソースにアクセスするためのアクセス許可をプリンシパルエンティティ (ユーザーまたはロール) に付与する必要があります。IAM 管理者は、アイデンティティベースのポリシーを エンティティにアタッチすることで権限を付与します。ただし、リソースベースのポリシーで、同じアカウントのプリンシパルへのアクセス権が付与されている場合は、アイデンティティベースのポリシーをさらに付与する必要はありません。詳細については、「IAM ユーザーガイド」の「IAM でのクロスアカウントリソースアクセス」を参照してください。

FreeRTOS のポリシーアクション

ポリシーアクションのサポート:あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

JSON ポリシーの `Action` 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連付けられた AWS API オペレーションと同じです。一致する API オペレーションのない許可のみのアクションなど、いくつかの例外があります。また、ポリシーに複数のアクションが必要なオペレーションもあります。これらの追加アクションは依存アクションと呼ばれます。

このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

FreeRTOS アクションのリストを確認するには、「サービス認可リファレンス」のFreeRTOS で定義されるアクション」を参照してください。

FreeRTOS のポリシーアクションは、アクションの前に以下のプレフィックスを使用します。

```
awes
```

単一のステートメントで複数のアクションを指定するには、アクションをカンマで区切ります。

```
"Action": [
      "awes:action1",
      "awes:action2"
         ]
```

FreeRTOS のアイデンティティベースのポリシーの例を確認するには、「FreeRTOS のアイデンティティベースのポリシーの例」を参照してください。

FreeRTOS のポリシーリソース

ポリシーリソースのサポート: あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

`Resource` JSON ポリシー要素はアクションが適用されるオブジェクトを指定します。ステートメントには`Resource` または `NotResource` 要素を含める必要があります。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。これは、リソースレベルの許可と呼ばれる特定のリソースタイプをサポートするアクションに対して実行できます。

オペレーションのリスト化など、リソースレベルの権限をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

FreeRTOS リソースタイプとその ARNsFreeRTOS で定義されるリソース」を参照してください。各リソースの ARN を指定できるアクションについては、「FreeRTOS で定義されるアクション」を参照してください。

FreeRTOS のアイデンティティベースのポリシーの例を確認するには、「FreeRTOS のアイデンティティベースのポリシーの例」を参照してください。

FreeRTOS のポリシー条件キー

サービス固有のポリシー条件キーのサポート: あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Condition 要素 (または Condition ブロック) を使用すると、ステートメントが有効な条件を指定できます。Condition 要素はオプションです。イコールや未満などの 条件演算子 を使用して条件式を作成して、ポリシーの条件とリクエスト内の値を一致させることができます。

1 つのステートメントに複数の Condition 要素を指定する場合、または 1 つの Condition 要素に複数のキーを指定する場合、 AWS では AND 論理演算子を使用してそれらを評価します。1 つの条件キーに複数の値を指定すると、 は論理ORオペレーションを使用して条件 AWS を評価します。ステートメントの権限が付与される前にすべての条件が満たされる必要があります。

条件を指定する際にプレースホルダー変数も使用できます。例えば IAM ユーザーに、IAM ユーザー名がタグ付けされている場合のみリソースにアクセスできる権限を付与することができます。詳細については、「IAM ユーザーガイド」の「IAM ポリシーの要素: 変数およびタグ」を参照してください。

AWS は、グローバル条件キーとサービス固有の条件キーをサポートしています。すべての AWS グローバル条件キーを確認するには、「IAM ユーザーガイド」のAWS 「グローバル条件コンテキストキー」を参照してください。

FreeRTOS 条件キーのリストを確認するには、「サービス認可リファレンス」のFreeRTOS の条件キー」を参照してください。条件キーを使用できるアクションおよびリソースについては、「FreeRTOS で定義されるアクション」を参照してください。

FreeRTOS のアイデンティティベースのポリシーの例を確認するには、「FreeRTOS のアイデンティティベースのポリシーの例」を参照してください。

FreeRTOS の ACL

ACL のサポート: なし

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

FreeRTOS での ABAC

ABAC (ポリシー内のタグ) のサポート: 一部

属性ベースのアクセス制御 (ABAC) は、属性に基づいてアクセス許可を定義する認可戦略です。では AWS、これらの属性はタグと呼ばれます。IAM エンティティ (ユーザーまたはロール) と多くの AWS リソースにタグをアタッチできます。エンティティとリソースのタグ付けは、ABAC の最初の手順です。その後、プリンシパルのタグがアクセスしようとしているリソースのタグと一致した場合にオペレーションを許可するように ABAC ポリシーをします。

ABAC は、急成長する環境やポリシー管理が煩雑になる状況で役立ちます。

タグに基づいてアクセスを管理するには、`aws:ResourceTag/`*`key-name`*、`aws:RequestTag/`*`key-name`*、または `aws:TagKeys` の条件キーを使用して、ポリシーの条件要素でタグ情報を提供します。

サービスがすべてのリソースタイプに対して 3 つの条件キーすべてをサポートする場合、そのサービスの値はありです。サービスが一部のリソースタイプに対してのみ 3 つの条件キーのすべてをサポートする場合、値は「部分的」になります。

ABAC の詳細については、「IAM ユーザーガイド」の「ABAC 認可でアクセス許可を定義する」を参照してください。ABAC をセットアップする手順を説明するチュートリアルについては、「IAM ユーザーガイド」の「属性ベースのアクセスコントロール (ABAC) を使用する」を参照してください。

FreeRTOS での一時的な認証情報の使用

一時的な認証情報のサポート: あり

一部の AWS のサービス は、一時的な認証情報を使用してサインインすると機能しません。一時的な認証情報 AWS のサービス を使用する などの詳細については、IAM ユーザーガイドの「IAM AWS のサービス と連携する 」を参照してください。

ユーザー名とパスワード以外の方法 AWS Management Console を使用して にサインインする場合、一時的な認証情報を使用します。たとえば、会社のシングルサインオン (SSO) リンク AWS を使用して にアクセスすると、そのプロセスによって一時的な認証情報が自動的に作成されます。また、ユーザーとしてコンソールにサインインしてからロールを切り替える場合も、一時的な認証情報が自動的に作成されます。ロールの切り替えに関する詳細については、「IAM ユーザーガイド」の「ユーザーから IAM ロールに切り替える (コンソール)」を参照してください。

一時的な認証情報は、 AWS CLI または AWS API を使用して手動で作成できます。その後、これらの一時的な認証情報を使用して アクセスできます AWS。長期的なアクセスキーを使用する代わりに、一時的な認証情報を動的に生成 AWS することをお勧めします。詳細については、「IAM の一時的セキュリティ認証情報」を参照してください。

FreeRTOS のクロスサービスプリンシパル許可

転送アクセスセッション (FAS) のサポート: あり

IAM ユーザーまたはロールを使用して でアクションを実行すると AWS、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、 を呼び出すプリンシパルのアクセス許可を AWS のサービス、ダウンストリームサービス AWS のサービス へのリクエストをリクエストする と組み合わせて使用します。FAS リクエストは、サービスが他の AWS のサービス またはリソースとのやり取りを完了する必要があるリクエストを受け取った場合にのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

FreeRTOS サービスロール

サービスロールのサポート: あり

サービスロールとは、サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、「IAM ユーザーガイド」の「AWS のサービスに許可を委任するロールを作成する」を参照してください。

> **Warning**
>
> サービスロールのアクセス許可を変更すると、FreeRTOS の機能が損なわれる可能性があります。FreeRTOS のガイダンスで指示されている場合を除き、サービスロールを編集しないでください。

FreeRTOS のサービスリンクロール

サービスにリンクされたロールのサポート: なし

サービスにリンクされたロールは、 にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント 、サービスによって所有されます。IAM 管理者は、サービスにリンクされたロールのアクセス許可を表示できますが、編集することはできません。

サービスにリンクされたロールの作成または管理の詳細については、「IAM と提携するAWS のサービス」を参照してください。表の「サービスリンクロール」列に Yes と記載されたサービスを見つけます。サービスにリンクされたロールに関するドキュメントをサービスで表示するには、[はい] リンクを選択します。

FreeRTOS のアイデンティティベースのポリシーの例

デフォルトでは、ユーザーおよびロールには、FreeRTOS リソースを作成または変更するアクセス許可はありません。また、、 AWS Command Line Interface （ AWS CLI ） AWS Management Console、または AWS API を使用してタスクを実行することはできません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き継ぐことができます。

これらサンプルの JSON ポリシードキュメントを使用して、IAM アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「IAM ポリシーを作成する (コンソール)」を参照してください。

各リソースタイプの ARN の形式など、FreeRTOS で定義されるアクションとリソースタイプの詳細については、「サービス認可リファレンス」のFreeRTOS のアクション、リソース、および条件キー」を参照してください。 ARNs

トピック

- ポリシーに関するベストプラクティス
- FreeRTOS コンソールの使用
- 自分の権限の表示をユーザーに許可する

ポリシーに関するベストプラクティス

アイデンティティベースのポリシーは、アカウント内でいずれかのユーザーが FreeRTOS リソースを作成、アクセス、または削除できるどうかを決定します。これらのアクションを実行すると、AWS アカウントに料金が発生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS 管理ポリシーを開始し、最小特権のアクセス許可に移行する – ユーザーとワークロードにアクセス許可の付与を開始するには、多くの一般的なユースケースにアクセス許可を付与するAWS 管理ポリシーを使用します。これらは で使用できます AWS アカウント。ユースケースに固有の AWS カスタマー管理ポリシーを定義することで、アクセス許可をさらに減らすことをお勧めしま

す。詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」または「ジョブ機能のAWS マネージドポリシー」を参照してください。

- 最小特権を適用する – IAM ポリシーで許可を設定する場合は、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、「IAM ユーザーガイド」の「IAM でのポリシーとアクセス許可」を参照してください。
- IAM ポリシーで条件を使用してアクセスをさらに制限する - ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。例えば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。条件を使用して、サービスアクションが などの特定の を通じて使用されている場合に AWS のサービス、サービスアクションへのアクセスを許可することもできます AWS CloudFormation。詳細については、「IAM ユーザーガイド」の「IAM JSON ポリシー要素:条件」を参照してください。
- IAM Access Analyzer を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する - IAM Access Analyzer は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM アクセスアナライザーは 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーの作成をサポートします。詳細については、「IAM ユーザーガイド」の「IAM Access Analyzer でポリシーを検証する」を参照してください。
- 多要素認証 (MFA) を要求する – で IAM ユーザーまたはルートユーザーを必要とするシナリオがある場合は AWS アカウント、セキュリティを強化するために MFA を有効にします。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、「IAM ユーザーガイド」の「MFA を使用した安全な API アクセス」を参照してください。

IAM でのベストプラクティスの詳細については、「IAM ユーザーガイド」の「IAM でのセキュリティのベストプラクティス」を参照してください。

FreeRTOS コンソールの使用

FreeRTOS コンソールにアクセスするには、最小限のアクセス許可のセットが必要です。これらのアクセス許可により、 の FreeRTOS リソースの詳細を一覧表示および表示できます AWS アカウント。最小限必要な許可よりも制限が厳しいアイデンティティベースのポリシーを作成すると、そのポリシーを持つエンティティ (ユーザーまたはロール) に対してコンソールが意図したとおりに機能しません。

AWS CLI または AWS API のみを呼び出すユーザーには、最小限のコンソールアクセス許可を付与する必要はありません。代わりに、実行しようとしている API オペレーションに一致するアクションのみへのアクセスが許可されます。

ユーザーとロールが引き続き FreeRTOS コンソールを使用できるようにするには、FreeRTOS *ConsoleAccess*または *ReadOnly* AWS 管理ポリシーもエンティティにアタッチします。詳細については、「IAM ユーザーガイド」の「ユーザーへのアクセス許可の追加」を参照してください。

自分の権限の表示をユーザーに許可する

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI または AWS API を使用してプログラムでこのアクションを実行するアクセス許可が含まれています。

```
{
    "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": "*"
        }
    ]
}
```

FreeRTOS ID とアクセスのトラブルシューティング

以下の情報は、FreeRTOS と IAM を使用する際に発生する可能性がある一般的な問題の診断や修正に役立ちます。

トピック

- FreeRTOS でアクションを実行する権限がない
- iam:PassRole を実行する権限がありません
- 自分の 以外のユーザーに FreeRTOS リソース AWS アカウント へのアクセスを許可する

FreeRTOS でアクションを実行する権限がない

アクションを実行する権限がないというエラーが表示された場合は、そのアクションを実行できるようにポリシーを更新する必要があります。

次のエラー例は、mateojackson IAM ユーザーがコンソールを使用して、ある *my-example-widget* リソースに関する詳細情報を表示しようとしたことを想定して、その際に必要なawes:*GetWidget* アクセス許可を持っていない場合に発生するものです。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes:GetWidget on resource: my-example-widget
```

この場合、awes:*GetWidget* アクションを使用して *my-example-widget*リソースへのアクセスを許可するように、mateojackson ユーザーのポリシーを更新する必要があります。

サポートが必要な場合は、 AWS 管理者にお問い合わせください。サインイン認証情報を提供した担当者が管理者です。

iam:PassRole を実行する権限がありません

iam:PassRole アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して FreeRTOS にロールを渡すことができるようにする必要があります。

一部の AWS のサービス では、新しいサービスロールまたはサービスにリンクされたロールを作成する代わりに、既存のロールをそのサービスに渡すことができます。そのためには、サービスにロールを渡す権限が必要です。

以下の例のエラーは、marymajor という IAM ユーザーがコンソールを使用して FreeRTOS でアクションを実行しようとした場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。メアリーには、ロールをサービスに渡す許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、Mary のポリシーを更新してメアリーに iam:PassRole アクションの実行を許可する必要があります。

サポートが必要な場合は、 AWS 管理者にお問い合わせください。サインイン資格情報を提供した担当者が管理者です。

自分の 以外のユーザーに FreeRTOS リソース AWS アカウント へのアクセスを許可する

他のアカウントのユーザーや組織外の人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまたはアクセスコントロールリスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください:

- FreeRTOS がこれらの機能をサポートしているかどうかを確認するには、「FreeRTOS と IAM の連携の仕組み」を参照してください。
- 所有 AWS アカウント している のリソースへのアクセスを提供する方法については、「IAM ユーザーガイド」の「所有 AWS アカウント している別の の IAM ユーザーへのアクセスを提供する」を参照してください。
- リソースへのアクセスをサードパーティーに提供する方法については AWS アカウント、IAM ユーザーガイドの「サードパーティー AWS アカウント が所有する へのアクセスを提供する」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、「IAM ユーザーガイド」の「外部で認証されたユーザー (ID フェデレーション) へのアクセスの許可」を参照してください。

- クロスアカウントアクセスにおけるロールとリソースベースのポリシーの使用方法の違いについては、「IAM ユーザーガイド」の「IAM でのクロスアカウントのリソースへのアクセス」を参照してください。

コンプライアンス検証

AWS のサービス が特定のコンプライアンスプログラムの範囲内にあるかどうかを確認するには、AWS のサービス「コンプライアンスプログラムによる範囲内」を参照して、関心のあるコンプライアンスプログラムを選択します。一般的な情報については、AWS 「 Compliance ProgramsAssurance」を参照してください。

を使用して、サードパーティーの監査レポートをダウンロードできます AWS Artifact。詳細については、「Downloading Reports in AWS Artifact」を参照してください。

を使用する際のお客様のコンプライアンス責任 AWS のサービス は、お客様のデータの機密性、貴社のコンプライアンス目的、適用される法律および規制によって決まります。 は、コンプライアンスに役立つ以下のリソース AWS を提供します。

- セキュリティのコンプライアンスとガバナンス – これらのソリューション実装ガイドでは、アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスの機能をデプロイする手順を示します。
- HIPAA 対応サービスのリファレンス – HIPAA 対応サービスの一覧が提供されています。すべての AWS のサービス が HIPAA の対象となるわけではありません。
- AWS コンプライアンスリソース – このワークブックとガイドのコレクションは、お客様の業界や地域に適用される場合があります。
- AWS カスタマーコンプライアンスガイド – コンプライアンスの観点から責任共有モデルを理解します。このガイドでは、複数のフレームワーク (米国国立標準技術研究所 (NIST)、Payment Card Industry Security Standards Council (PCI)、国際標準化機構 (ISO) など) にわたるセキュリティコントロールを保護および AWS のサービス マッピングするためのベストプラクティスをまとめています。
- 「 デベロッパーガイド」の「ルールによるリソースの評価」 – この AWS Config サービスは、リソース設定が内部プラクティス、業界ガイドライン、および規制にどの程度準拠しているかを評価します。 AWS Config
- AWS Security Hub – これにより AWS のサービス 、 内のセキュリティ状態を包括的に把握できます AWS。Security Hub では、セキュリティコントロールを使用して AWS リソースを評価し、セキュリティ業界標準とベストプラクティスに対するコンプライアンスをチェックします。サポー

トされているサービスとコントロールの一覧については、Security Hub のコントロールリファレンスを参照してください。

- Amazon GuardDuty – 不審なアクティビティや悪意のあるアクティビティがないか環境をモニタリングすることで AWS アカウント、、ワークロード、コンテナ、データに対する潜在的な脅威 AWS のサービス を検出します。GuardDuty を使用すると、特定のコンプライアンスフレームワークで義務付けられている侵入検知要件を満たすことで、PCI DSS などのさまざまなコンプライアンス要件に対応できます。
- AWS Audit Manager – これにより AWS のサービス 、 AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

の耐障害性 AWS

AWS グローバルインフラストラクチャは、 AWS リージョンとアベイラビリティーゾーンを中心に構築されています。 AWS リージョンには、低レイテンシー、高いスループット、そして高度の冗長ネットワークで接続されている複数の物理的に独立し隔離されたアベイラビリティーゾーンがあります。アベイラビリティーゾーンでは、アベイラビリティーゾーン間で中断せずに、自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、およびスケーラビリティが優れています。

AWS リージョンとアベイラビリティーゾーンの詳細については、AWS 「グローバルインフラストラクチャ」を参照してください。

FreeRTOS でのインフラストラクチャセキュリティ

AWS マネージドサービスは、ホワイトペーパー「Amazon Web Services: セキュリティプロセスの概要」に記載されている AWS グローバルネットワークセキュリティ手順で保護されています。

AWS 公開された API コールを使用して、ネットワーク経由で AWS サービスにアクセスします。クライアントは、Transport Layer Security (TLS) 1.2 以降をサポートする必要があります。TLS 1.3 以降が推奨されます。また、一時的ディフィー・ヘルマン Ephemeral Diffie-Hellman (DHE) や Elliptic Curve Ephemeral Diffie-Hellman (ECDHE) などの Perfect Forward Secrecy (PFS) を使用した暗号スイートもクライアントでサポートされている必要があります。これらのモードは、Java 7 以降など、最近のほとんどのシステムでサポートされています。

また、リクエストにはアクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。または、AWS Security Token ServiceAWS STSを使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

Amazon FreeRTOS Github リポジトリ移行ガイド

現在非推奨の Amazon FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトがある場合は、次の手順に従ってください。

1. 公開されている最新のセキュリティ修正を使用して最新の状態に保ちます。「FreeRTOS LTS libraries」のページで最新情報を確認するか、FreeRTOS-LTS GitHub リポジトリに登録して、重大なバグ修正とセキュリティバグ修正を含む最新の LTS パッチを入手してください。必要な最新の FreeRTOS LTS パッチは、個々の GitHub リポジトリから直接ダウンロードまたは複製できます。
2. ネットワークトランスポートインターフェイスの実装をリファクタリングして、ハードウェアプラットフォームを最適化することを検討してください。セキュアソケットや Wifi API のような抽象化 API は、最新の coreMQTT ライブラリでは必要ありません。詳細については、「Transport Interface」を参照してください。

付録

次の表は、Amazon FreeRTOS リポジトリ内のすべてのデモプロジェクト、レガシーライブラリ、および抽象化 API の推奨事項を示しています。

移行されたライブラリとデモ

名前	型	レコメンデーション	
coreHTTP	デモとライブラリ	FreeRTOS Github organization の coreHTTP リポジトリ (git を使用している場合はサブモジュール) から coreHTTP ライブラリを直接複製またはダウンロードします。coreHTTP デモはプライマリ FreeRTOS ディストリビューションに含まれています。詳細については、coreHTTP のページを参照してください。	
coreMQTT	デモとライブラリ	FreeRTOS Github organization の coreMQTT リポジトリ (git を使用している場合はサブモジュール)	

名前	型	レコメンデーション	
		から coreMQTT ライブラリを直接複製またはダウンロードします。coreMQTT デモはプライマリ FreeRTOS ディストリビューションに含まれています。詳細については、coreMQTT のページを参照してください。	
coreMQTT-Agent	デモとライブラリ	FreeRTOS Github organization の coreMQTT-Agent リポジトリ (git を使用している場合はサブモジュール) から coreMQTT-Agent ライブラリを直接複製またはダウンロードします。coreMQTT-Agent のデモは coreMQTT-Agent-Demos リポジトリにあります。詳細については、coreMQTT-Agent のページを参照してください。	
device_defender_for_aws	デモとライブラリ	AWS IoT Device Defender ライブラリは AWS GitHub 組織のリポジトリにあります。AWS IoT Device Defender リポジトリから直接 (git を使用している場合はサブモジュールを) 複製またはダウンロードします。 AWS IoT Device Defender デモは、プライマリ FreeRTOS ディストリビューションにあります。詳細については、AWS IoT Device Defender のページを参照してください。	

名前	型	レコメンデーション	
device_shadow_for_aws	デモとライブラリ	AWS IoT Device Shadow ライブラリは、AWS GitHub 組織のリポジトリにあります。AWS IoT Device Shadow リポジトリから直接 (git を使用している場合はサブモジュールを) 複製またはダウンロードします。 AWS IoT Device Shadow デモは、プライマリ FreeRTOS ディストリビューションにあります。詳細については、AWS IoT Device Shadow のページを参照してください。	
jobs_for_aws	デモとライブラリ	AWS IoT Jobs ライブラリは AWS GitHub 組織のリポジトリにあります。AWS IoT Jobs リポジトリから直接 (git を使用している場合はサブモジュールを) 複製またはダウンロードします。 AWS IoT ジョブデモは、プライマリ FreeRTOS ディストリビューションにあります。詳細については、AWS IoT Jobs のページを参照してください。	
OTA	デモとライブラリ	AWS IoT Over-The-Air (OTA) 更新ライブラリは、AWS GitHub 組織のリポジトリにあります。AWS IoT OTA リポジトリから直接 (git を使用している場合はサブモジュールを) 複製またはダウンロードします。 AWS IoT OTA デモはプライマリ FreeRTOS ディストリビューションにあります。詳細については、AWS IoT OTA のページを参照してください。	

名前	型	レコメンデーション	
CLI と FreeRTOS_Plus_CLI	デモとライブラリ	WinSim で実行されている CLI のサンプルがあります。詳細については、FreeRTOS Plus コマンドラインインターフェイスのページを参照してください。NXP i.MX RT1060 および STM32U5 プラットフォームでの注目の FreeRTOS IoT リファレンス統合では、実際のハードウェアでの CLI サンプルも提供しています。	
ログ記録	マクロ	一部の FreeRTOS ライブラリでは、特定のハードウェアプラットフォーム用のロギングマクロが実装されています。ロギングマクロの実装方法については、ロギングのページを参照してください。実際のハードウェアでの実行例については、FreeRTOS の IoT リファレンスのいずれかを参照してください。	
greengrass_connectivity	デモ	〔移行中] このデモプロジェクトでは、 AWS IoT Greengrass デバイスに接続する前にクラウド接続が利用可能であることを前提としています。ローカル認証と検出機能を実証する新しいプロジェクトが開発中です。新しいデモプロジェクトは、間もなく FreeRTOS Github organization で公開される予定です。	

非推奨ライブラリとデモ

名前	型	レコメンデーション
BLE	デモとライブラリ	FreeRTOS BLE ライブラリは、独自の MQTT プロトコルを実装し、携帯電話などのプロキシデバイスを使用して、Bluetooth Low Energy (BLE) を介して MQTT トピックの公開およびサブスクライブをサポートします。これはもはや義務付けられているものではありません。独自の BLE スタックか、Nimble などのサードパーティオプションを使用して、プロジェクトを最適化してください。
dev_mode_key_provisioning	デモ	NXP i.MX RT1060、STM32U5、または ESP32-C3 プラットフォームでの注目の FreeRTOS IoT リファレンス統合には、CLI を使用した重要なプロビジョニングの例が示されています。
posix	抽象化とデモ	使用はお勧めしません。
wifi_provisioning	例	この例では、Amazon FreeRTOS BLE ライブラリを使用してデバイスに WiFi 認証情報をプロビジョニングする方法を示しました。BLE による WiFi プロビジョニングの例については、ESP32C3 プラットフォームの FreeRTOS 特集 IoT リファレンスを参照してください。
レガシー抽象化 API	コード	これらは、さまざまなベンダーのさまざまなサードパーティ製ソフトウェアスタック、接続モジュール、MCU プラットフォームに抽象

名前	型	レコメンデーション	
		化インターフェイスを提供するために作成された API です。例えば、Wi Fi 抽象化やセキュアソケットなどのインターフェイスがあります。これらは Amazon FreeRTOS リポジトリでサポートされており、フォルダ `/libraries/abstractions/` にあります。FreeRTOS LTS ライブラリを使用する場合、これらの API は必要ありません。	

上の表のライブラリとデモには、セキュリティパッチやバグ修正は適用されません。

サードパーティライブラリ

Amazon FreeRTOS のデモでサードパーティのライブラリを使用する場合は、サードパーティのリポジトリから直接サブモジュール化することをお勧めします。

- CMock: Cmock リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。
- jsmn: 推奨されておらず、サポートも終了しています。
- lwip: lwip-tcpip リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。
- lwip_osal: お使いのハードウェアプラットフォーム/ボードに lwip_osal を実装する方法については、i.MX RT1060 または STM32U5 で FreeRTOS 特集リファレンス統合を参照してください。
- mbedtls: Mbed-TLS リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。mbedtls の設定とユーティリティは再利用できます。この場合はローカルコピーを作成してください。
- pkcs11: corePKCS11 ライブラリまたは OASIS PKCS 11 リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。
- tinycbor: tinycbor リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。
- tinycrypt: 使用している MCU プラットフォームの暗号アクセラレーターを使用することを推奨します (可能な場合)。tinycrypt を引き続き使用したい場合は、tinycrypt リポジトリから直接 (git を使っている場合はサブモジュールを) 複製してください。
- tracealyzer_recorder: Percepio の trace recorder リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。

- unity: ThrowTheSwitch/Unity リポジトリから直接 (git を使っている場合はサブモジュールを) 複製します。
- win_pcap: win_pcap のメンテナンスは行われなくなりました。代わりに libslirp、libpcap (posix)、または npcap を使用することをお勧めします。

移植テストと統合テスト

FreeRTOS ライブラリの統合を検証するために必要な `/tests` フォルダ内のすべてのテストは、FreeRTOS-Libraries-Integration-Tests リポジトリに移行されました。これらは PAL の実装とライブラリ統合のテストに使用できます。Device Tester (IDT) では、FreeRTOS AWS IoT の Device Qualification Program にも同じテストが使用されます。 AWS FreeRTOS

アーカイブされた FreeRTOS ドキュメント

このコンテンツはアーカイブされ、更新されなくなりました。古い情報や廃止された機能を参照できます。FreeRTOS に関する最新のガイダンスについては、この FreeRTOS ユーザーガイドまたはその他の関連 AWS ドキュメントの最新情報を参照してください。最新情報については、「」を参照してくださいFreeRTOS とは。

トピック

- FreeRTOS ユーザーガイドのアーカイブ
- 以前の「FreeRTOS ユーザーガイド」のコンテンツ

FreeRTOS ユーザーガイドのアーカイブ

ここにある以前のバージョンの FreeRTOS ユーザーガイドは、FreeRTOS LTS (長期サポート) リリースがあれば使用できます。

- FreeRTOS バージョン 202210.00 用の「FreeRTOS ユーザーガイド」
- バージョン 202012.00 用の FreeRTOS ユーザーガイド

以前の「FreeRTOS ユーザーガイド」のコンテンツ

このコンテンツは廃止されましたが、参照としてここに記載されています。

最近のコンテンツへのリンクについては、FreeRTOS の使用を開始する を参照してください。

FreeRTOS の使用を開始する

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

FreeRTOS の開始方法チュートリアルは、ホストマシンで FreeRTOS をダウンロードして設定してから、認定されたマイクロコントローラーボードでシンプルなデモアプリケーションをコンパイルして実行する方法について説明します。

このチュートリアルでは、 AWS IoT と AWS IoT コンソールに精通していることを前提としています。そうでない場合は、先に進む前に AWS IoT の使用開始のチュートリアルを完了することをお勧めします。

トピック:

- FreeRTOS デモアプリケーション
- 最初のステップ
- トラブルシューティングの開始方法
- FreeRTOS で CMake を使用する
- 開発者モードのキーのプロビジョニング
- ボード固有の入門ガイド
- FreeRTOS での次のステップ

FreeRTOS デモアプリケーション

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルのデモアプリケーションは、*freertos*/demos/coreMQTT_Agent/mqtt_agent_task.c ファイルに定義されている coreMQTT エージェントデモです。を使用して AWS クラウドcoreMQTT ライブラリに接続し、MQTT AWS IoT ブローカーがホストする MQTT トピックに定期的にメッセージを発行します。

一度に実行できるのは 1 つの FreeRTOS デモアプリケーションのみです。FreeRTOS デモプロジェクトを構築する場合、*freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h ヘッダーファイルで有効になっている最初のデモが、実行

されるアプリケーションです。このチュートリアルでは、いずれのデモも有効または無効にする必要はありません。coreMQTT エージェントデモは、デフォルトで有効になっています。

FreeRTOS に付属するデモアプリケーションの詳細については、「FreeRTOS デモ」を参照してください。

最初のステップ

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

で FreeRTOS の使用を開始するには AWS IoT、 AWS アカウント、 へのアクセス権限を持つユーザー AWS IoT 、および FreeRTOS クラウドサービスが必要です。また、FreeRTOS をダウンロードし、ボードの FreeRTOS デモプロジェクトが動作するように設定する必要があります AWS IoT。以下のセクションでは、これらの要件について説明します。

> **Note**
>
> - ESP32-DevKitC、ESP-WROVER-KIT、または ESP32-WROOM-32SE を使用している場合は、これらの最初のステップをスキップして Espressif ESP32-DevKitC と ESP-WROVER-KIT の開始方法 に進みます。
> - Nordic nRF52840-DK を使用している場合は、これらの最初のステップをスキップして「Nordic nRF52840-DK の開始方法」に進みます。

1. AWS アカウントとアクセス許可の設定
2. への MCU ボードの登録 AWS IoT
3. FreeRTOS をダウンロードする
4. FreeRTOS デモを設定する

AWS アカウントとアクセス許可の設定

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

手順については、IAM ユーザーガイドの AWS アカウント「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、AWS IAM Identity Center 「 ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン 「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

  アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。
- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。
- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

への MCU ボードの登録 AWS IoT

AWS クラウドと通信 AWS IoT するには、ボードを に登録する必要があります。ボードを に登録するには AWS IoT、以下が必要です。

AWS IoT ポリシー

この AWS IoT ポリシーは、 AWS IoT リソースにアクセスするためのアクセス許可をデバイスに付与します。クラウドに保存されます AWS 。

AWS IoT モノ

AWS IoT モノを使用すると、 でデバイスを管理できます AWS IoT。クラウドに保存されます AWS 。

プライベートキーと X.509 証明書

プライベートキーと証明書により、デバイスは で認証できます AWS IoT。

ボードを登録するには、次の手順に従ってください。

AWS IoT ポリシーを作成するには

1. IAM ポリシーを作成するには、 AWS リージョンと AWS アカウント番号を知っている必要があります。

AWS アカウント番号を検索するには、 AWS マネジメントコンソールを開き、右上隅のアカウント名の下にあるメニューを見つけて展開し、マイアカウントを選択します。アカウント ID が [Account Settings] (アカウント設定) に表示されます。

AWS アカウントの AWS リージョンを検索するには、 を使用します AWS Command Line Interface。をインストールするには AWS CLI、 AWS Command Line Interface ユーザーガイドの手順に従ってください。をインストールしたら AWS CLI、コマンドプロンプトウィンドウを開き、次のコマンドを入力します。

```
aws iot describe-endpoint --endpoint-type=iot:Data-ATS
```

出力は次のようになります:

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

この例では、リージョンは `us-west-2` です。

> **Note**
>
> 例に示されているように、ATS エンドポイントを使用することをお勧めします。

2. AWS IoT コンソールを参照します。
3. ナビゲーションペインで、[Secure] (保護) を選択し、[Policies] (ポリシー) を選択してから [Create] (作成) を選択します。
4. ポリシーを識別するための名前を入力します。
5. [Add statements] (ステートメントを追加) セクションで、[Advanced mode] (アドバンストモード) を選択します。次の JSON をポリシーエディタウィンドウにコピーして貼り付けます。*aws-region* と を AWS リージョンとアカウント ID *aws-account* に置き換えます。

```
{
    "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:*"
    }
    ]
}
```

このポリシーは以下のアクセス権限を与えます。

`iot:Connect`

任意のクライアント ID を使用して AWS IoT メッセージブローカーに接続するアクセス許可をデバイスに付与します。

`iot:Publish`

MQTT トピックについて MQTT メッセージを発行するためのアクセス許可をデバイスに付与します。

`iot:Subscribe`

MQTT トピックフィルターをサブスクライブするためのアクセス許可をデバイスに付与します。

`iot:Receive`

MQTT トピックについて AWS IoT メッセージブローカーからメッセージを受信するためのアクセス許可をデバイスに付与します。

6. [Create] (作成) を選択します。

デバイス用の IoT モノ、プライベートキー、証明書を作成するには

1. AWS IoT コンソールを参照します。
2. ナビゲーションペインで、[Manage] (管理)、[Things] (モノ) の順に選択します。
3. アカウントに IoT モノが登録されていない場合は、[You don't have any things yet] (まだモノがありません) ページが表示されます。このページが表示された場合は、[Register a thing] (モノの登録) を選択します。それ以外の場合は、[Create] (作成) を選択します。
4. AWS IoT モノの作成ページで、モノを 1 つ作成する を選択します。
5. [Add your device to the thing registry] (Thing Registry にデバイスを追加) ページで、モノの名前を入力してから [Next] (次へ) を選択します。
6. [Add a certificate for your thing] (モノに証明書を追加) ページの [One-click certificate creation] (1-Click 証明書作成) から [Create certificate] (証明書の作成) を選択します。
7. それぞれの [Download] (ダウンロード) リンクを選択して、プライベートキーと証明書をダウンロードします。
8. 証明書を有効にするには、[Activate] (有効化) を選択します。証明書は、使用前にアクティブ化する必要があります。
9. ポリシーをアタッチを選択して、デバイスに AWS IoT オペレーションへのアクセスを許可するポリシーを証明書にアタッチします。
10. 作成したポリシーを選択し、[Register thing] (モノの登録) を選択します。

ボードが に登録されたら AWS IoT、 に進むことができますFreeRTOS をダウンロードする。

FreeRTOS をダウンロードする

FreeRTOS は、FreeRTOS GitHub リポジトリからダウンロードできます。

FreeRTOS をダウンロードしたら、FreeRTOS デモを設定する の手順を実行できます。

FreeRTOS デモを設定する

ボードでデモをコンパイルして実行するには、FreeRTOS ディレクトリで一部の設定ファイルを編集する必要があります。

AWS IoT エンドポイントを設定するには

ボードで実行されているアプリケーションが正しい AWS IoT エンドポイントにリクエストを送信できるように、エンドポイントに FreeRTOS を指定する必要があります。

1. AWS IoT コンソールを参照します。
2. 左側のナビゲーションペインで [設定] を選択します。

   AWS IoT エンドポイントはデバイスデータエンドポイントに表示されます。次のようになっているはずです。*1234567890123*-ats.iot.*us-east-1*.amazonaws.comこのエンドポイントを書きとめておきます。
3. ナビゲーションペインで、[Manage] (管理)、[Things] (モノ) の順に選択します。

   デバイスには AWS IoT モノの名前が必要です。この名前を書き留めておきます。
4. demos/include/aws_clientcredential.h を開きます。
5. 以下の定数に値を指定します。

   - #define clientcredentialMQTT_BROKER_ENDPOINT "*Your AWS IoT endpoint*";
   - #define clientcredentialIOT_THING_NAME "*The AWS IoT thing name of your board*"

Wi-Fi を設定するには

ボードが Wi-Fi 接続経由でインターネットに接続している場合は、ネットワークに接続するための Wi-Fi 認証情報を FreeRTOS に渡す必要があります。Wi-Fi がボードでサポートされていない場合は、以下のステップをスキップできます。

1. demos/include/aws_clientcredential.h.
2. 以下の #define 定数の値を指定します。

   - #define clientcredentialWIFI_SSID "*The SSID for your Wi-Fi network*"
   - #define clientcredentialWIFI_PASSWORD "*The password for your Wi-Fi network*"
   - #define clientcredentialWIFI_SECURITY *Wi-Fi ###############*

     有効なセキュリティタイプは以下の通りです。

     - eWiFiSecurityOpen (オープン、セキュリティなし)
     - eWiFiSecurityWEP (WEP セキュリティ)
     - eWiFiSecurityWPA (WPA セキュリティ)
     - eWiFiSecurityWPA2 (WPA2 セキュリティ)

AWS IoT 認証情報をフォーマットするには

FreeRTOS は、デバイス AWS IoT に代わって と正常に通信するために、登録済みモノとアクセス許可ポリシーに関連付けられた AWS IoT 証明書とプライベートキーを持っている必要があります。

> **Note**
>
> AWS IoT 認証情報を設定するには、デバイスを登録したときに AWS IoT コンソールからダウンロードしたプライベートキーと証明書が必要です。デバイスを AWS IoT モノとして登録したら、 AWS IoT コンソールからデバイス証明書を取得できますが、プライベートキーを取得することはできません。

FreeRTOS は C 言語のプロジェクトであり、証明書とプライベートキーをプロジェクトに追加するには、特別な形式にする必要があります。

1. ブラウザウィンドウで、`tools/certificate_configuration/CertificateConfigurator.html` を開きます。
2. [Certificate PEM file] (証明書 PEM ファイル) で、 AWS IoT コンソールからダウンロードした *`ID`*`-certificate.pem.crt` を選択します。
3. [Private Key PEM file] (プライベートキー PEM ファイル) で、 AWS IoT コンソールからダウンロードした *`ID`*`-private.pem.key` を選択します。
4. [Generate and save aws_clientcredential_keys.h] (aws_clientcredential_keys.h の生成と保存) を選択して、ファイルを `demos/include` に保存します。これにより、ディレクトリ内の既存ファイルが上書きされます。

   > **Note**
   >
   > 証明書とプライベートキーは、デモ専用にハードコードされています。本番稼動レベルのアプリケーションでは、これらのファイルを安全な場所に保存する必要があります。

FreeRTOS の設定を完了した後は、ボードの入門ガイドを参照し、プラットフォームのハードウェアおよびソフトウェア開発環境をセットアップして、ボードでデモをコンパイルして実行できます。ボード固有の説明については、「ボード固有の入門ガイド」を参照してください。入門ガイドチュートリアルで使用されているデモアプリケーションは、coreMQTT Mutual Authentication デモで、`demos/coreMQTT/mqtt_demo_mutual_auth.c` にあります。

トラブルシューティングの開始方法

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

以下のトピックは、FreeRTOS の使用開始時に発生した問題のトラブルシューティングに役立ちます。

トピック

- 一般的なトラブルシューティングの開始方法のヒント
- ターミナルエミュレーターをインストールする

ボード固有のトラブルシューティングについては、ボードの「FreeRTOS の使用を開始する」を参照してください。

一般的なトラブルシューティングの開始方法のヒント

Hello World デモプロジェクトを実行した後、 AWS IoT コンソールにメッセージは表示されません。何をすればよいですか?

次の操作を試してください：

1. ターミナルウィンドウを開き、サンプルのログ出力を表示します。これは何が間違っているのかを判断するのに役立ちます。
2. ネットワーク認証情報が有効であることを確認します。

デモの実行時にターミナルに表示されるログが切り捨てられます。表示幅を増やすにはどうすればいいですか?

実行しているデモの `FreeRTOSconfig.h` ファイルで `configLOGGING_MAX_MESSAGE_LENGTH` の値を 255 に増やします。

```
#define configLOGGING_MAX_MESSAGE_LENGTH    255
```

ターミナルエミュレーターをインストールする

ターミナルエミュレーターは、問題を診断したり、デバイスコードが正常に動作しているかを検証するのに役立ちます。Windows、macOS、および Linux で使用できる、さまざまなターミナルエミュレーターがあります。

ターミナルエミュレーターを使用してボードとのシリアル接続を確立する前に、ボードをコンピュータに接続する必要があります。

ターミナルエミュレーターを設定するには、次の設定を使用します。

ターミナルの設定	値
ボーレート	115200
データ	8 ビット
パリティ	なし
停止	1 ビット
フロー制御	なし

ボードのシリアルポートを見つける

ボードのシリアルポートが不明な場合は、コマンドラインまたはターミナルから次のいずれかのコマンドを発行して、ホストコンピュータに接続されているすべてのデバイスのシリアルポートを返すことができます。

Windows

```
chgport
```

リナックス

```
ls /dev/tty*
```

macOS

```
ls /dev/cu.*
```

FreeRTOS で CMake を使用する

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

CMake を使用して、FreeRTOS アプリケーションのソースコードからプロジェクトビルドファイルを生成し、ソースコードを構築および実行できます。

また、IDE を使用して、FreeRTOS 対応デバイスでコードの編集、デバッグ、コンパイル、フラッシュ、実行を行うこともできます。各ボード固有の入門ガイドには、特定のプラットフォームで IDE をセットアップするための手順が記載されています。IDE なしで作業する場合は、コードの開発とデバッグ用に他のサードパーティーのコード編集およびデバッグツールを使用してから、CMake を使用してアプリケーションをビルドして実行できます。

CMake をサポートしているボードは次のとおりです。

- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- Infineon XMC4800 IoT 接続キット
- Marvell MW320 AWS IoT Starter キット
- Marvell MW322 AWS IoT Starter キット
- Microchip Curiosity PIC32MZEF バンドル
- Nordic nRF52840 DK Development kit
- STMicroelectronicsSTM32L4 Discovery Kit IoT Node
- Texas Instruments CC3220SF-LAUNCHXL
- Microsoft Windows Simulator

FreeRTOS で CMake を使用する方法の詳細については、以下のトピックを参照してください 。

トピック

- 前提条件

- サードパーティーのコードエディタおよびデバッグツールによる FreeRTOS アプリケーションの開発
- CMake で FreeRTOS を構築する

前提条件

続行する前に、ホストマシンが次の前提条件を満たしていることを確認してください。

- デバイスのコンパイルツールチェーンは、マシンのオペレーティングシステムをサポートしている必要があります。CMake はWindows、macOS、Linux の全バージョンをサポートしています。

  Windows Subsystem for Linux (WSL) はサポートされていません。Windows マシンでネイティブ CMake を使用します。
- CMake バージョン 3.13 以降がインストールされている必要があります。

  CMake のバイナリディストリビューションは CMake.org からダウンロードできます。

  > **Note**
  >
  > CMake のバイナリディストリビューションをダウンロードする場合は、コマンドラインから CMake を使用する前に、必ず CMake 実行可能ファイルを PATH 環境変数に追加してください。

  macOS では Homebrew、Windows では scoop や chocolatey などのパッケージマネージャーを使って CMake をダウンロードしてインストールすることもできます。

  > **Note**
  >
  > 多くの Linux ディストリビューションのパッケージマネージャーで提供されている CMake パッケージのバージョンは古くなっています。ディストリビューションのパッケージマネージャーで CMake の最新バージョンが提供されていない場合は、`linuxbrew` または `nix` などの代替パッケージマネージャーを試すことができます。
- 互換性のあるネイティブビルドシステムが必要です。

  CMake は GNU Make や Ninja を含む多くのネイティブビルドシステムをターゲットにすることができます。Make と Ninja の両方とも、Linux、macOS、および Windows 上のパッケージマネー

ジャーでインストールできます。Windows で Make を使用している場合は、Equation からスタンドアロンバージョンをインストールすることも、Make をバンドルした MinGW をインストールすることもできます。

> **Note**
>
> MinGW の Make 実行可能ファイルは `make.exe` ではなく `mingw32-make.exe` と呼ばれます。

Make よりも高速で、すべてのデスクトップオペレーティングシステムにネイティブサポートを提供するため、Ninja を使用することをお勧めします。

サードパーティーのコードエディタおよびデバッグツールによる FreeRTOS アプリケーションの開発

コードエディタとデバッグ拡張機能またはサードパーティーのデバッグツールを使用して、FreeRTOS 用にアプリケーションを開発できます。

たとえば、コードエディタとして Visual Studio を使用すると、コードエディタを使用すると、デバッガーとして Cortex-Debug VS Code 拡張機能をインストールできます。アプリケーションの開発が完了したら、CMake コマンドラインツールを呼び出して VS Code 内からプロジェクトをビルドできます。CMake を使用して FreeRTOS アプリケーションを構築する方法の詳細については、「CMake で FreeRTOS を構築する」を参照してください。

デバッグの場合、以下のようなデバッグ設定で VS Code を指定できます。

```
"configurations": [
    {
        "name": "Cortex Debug",
        "cwd": "${workspaceRoot}",
        "executable": "./build/st/stm32l475_discovery/aws_demos.elf",
        "request": "launch",
        "type": "cortex-debug",
        "servertype": "stutil"
    }
]
```

CMake で FreeRTOS を構築する

CMake はデフォルトであなたのホストオペレーティングシステムをターゲットシステムとしてターゲットにします。クロスコンパイルにこれを使用するためには、CMake には使用するコンパイラを指定するツールチェーンファイルが必要です。FreeRTOS では、*freertos*/tools/cmake/toolchains にデフォルトのツールチェーンファイルが用意されています。CMake にこのファイルを提供する方法は、CMake コマンドラインインターフェイスあるいは GUI のどちらを使用しているかに応じて異なります。詳細については、次の「ビルドファイルの生成 (CMake コマンドラインツール)」の手順に従います。CMake のクロスコンパイルの詳細については、公式 CMake Wiki の「クロスコンパイル」を参照してください。

CMake ベースのプロジェクトを構築するには

1. CMake を実行して、Make や Ninja などのネイティブビルドシステムのビルドファイルを生成します。

   ネイティブビルドシステム用のビルドファイルを生成するには、CMakeコマンドラインツールまたは CMake GUI を使用できます。

   FreeRTOS ビルドファイルの生成については、「ビルドファイルの生成 (CMake コマンドラインツール)」および「ビルドファイルの生成 (CMake GUI)」を参照してください。
2. プロジェクトを実行可能ファイルにするには、ネイティブビルドシステムを起動します。

   FreeRTOS ビルドファイルの作成については、「生成されたビルドファイルから FreeRTOS を構築する」を参照してください。

ビルドファイルの生成 (CMake コマンドラインツール)

CMake コマンドラインツール (cmake) を使用して、FreeRTOS にビルドファイルを生成できます。ビルドファイルを生成するには、ターゲットボード、コンパイラ、およびソースコードの場所を指定してディレクトリを構築する必要があります。

cmake には次のオプションを使用できます。

- -DVENDOR – ターゲットボードを指定します。
- -DCOMPILER – コンパイラを指定します。
- -S: ソースコードの場所を指定します。
- -B – 生成されたビルドファイルの場所を指定します。

> **Note**
>
> コンパイラはシステムの PATH 変数内にあるか、コンパイラの場所を指定する必要があります。

たとえば、ベンダーが Texas Instruments、ボードが CC3220 Launchpad、コンパイラが GCC for ARM の場合、次のコマンドを発行して、現在のディレクトリから *build-directory* という名前のディレクトリにソースファイルを構築できます。

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

> **Note**
>
> Windows を使用している場合、CMake はデフォルトで Visual Studio を使用するため、ネイティブビルドシステムを指定する必要があります。例:
>
> ```
> cmake -DVENDOR=ti -DBOARD=cc3220_launchpad -DCOMPILER=arm-ti -S . -B build-directory -G Ninja
> ```
>
> または:
>
> ```
> cmake -DVENDOR=ti -DBOARD=cc3220_launchpad -DCOMPILER=arm-ti -S . -B build-directory -G "MinGW Makefiles"
> ```

正規表現 ${VENDOR}.* と ${BOARD}.* は、一致するボードを検索するために使用されるので、VENDOR と BOARD のオプションにベンダーとボードのフルネームを使用する必要はありません。部分一致は、単一の一致がある場合に限り機能します。たとえば、次のコマンドは同じソースから同じビルドファイルを生成します。

```
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
```

デフォルトのディレクトリ cmake/toolchains にないツールチェーンファイルを使用する場合は、CMAKE_TOOLCHAIN_FILE オプションを使用できます。例:

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

ツールチェーンファイルでコンパイラの絶対パスが使用されておらず、コンパイラを PATH 環境変数に追加しなかった場合、CMake はそれを見つけられない可能性があります。CMake がツールチェーンファイルを見つけられるように、AFR_TOOLCHAIN_PATH オプションを使用できます。このオプションは、指定されたツールチェーンディレクトリのパスと bin の下のツールチェーンのサブフォルダを検索します。例:

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

デバッグを有効にするには、CMAKE_BUILD_TYPE を debug に設定します。このオプションを有効にすると、CMake はコンパイルオプションにデバッグフラグを追加し、デバッグシンボルを使って FreeRTOS を構築します。

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

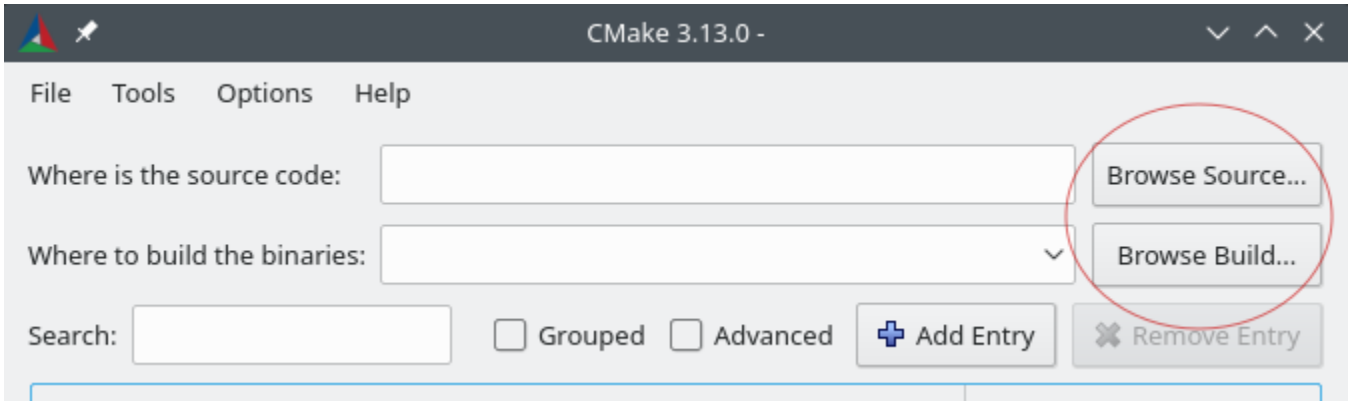
また、CMAKE_BUILD_TYPE を release に設定して、コンパイルオプションに最適化フラグを追加することもできます。

ビルドファイルの生成 (CMake GUI)

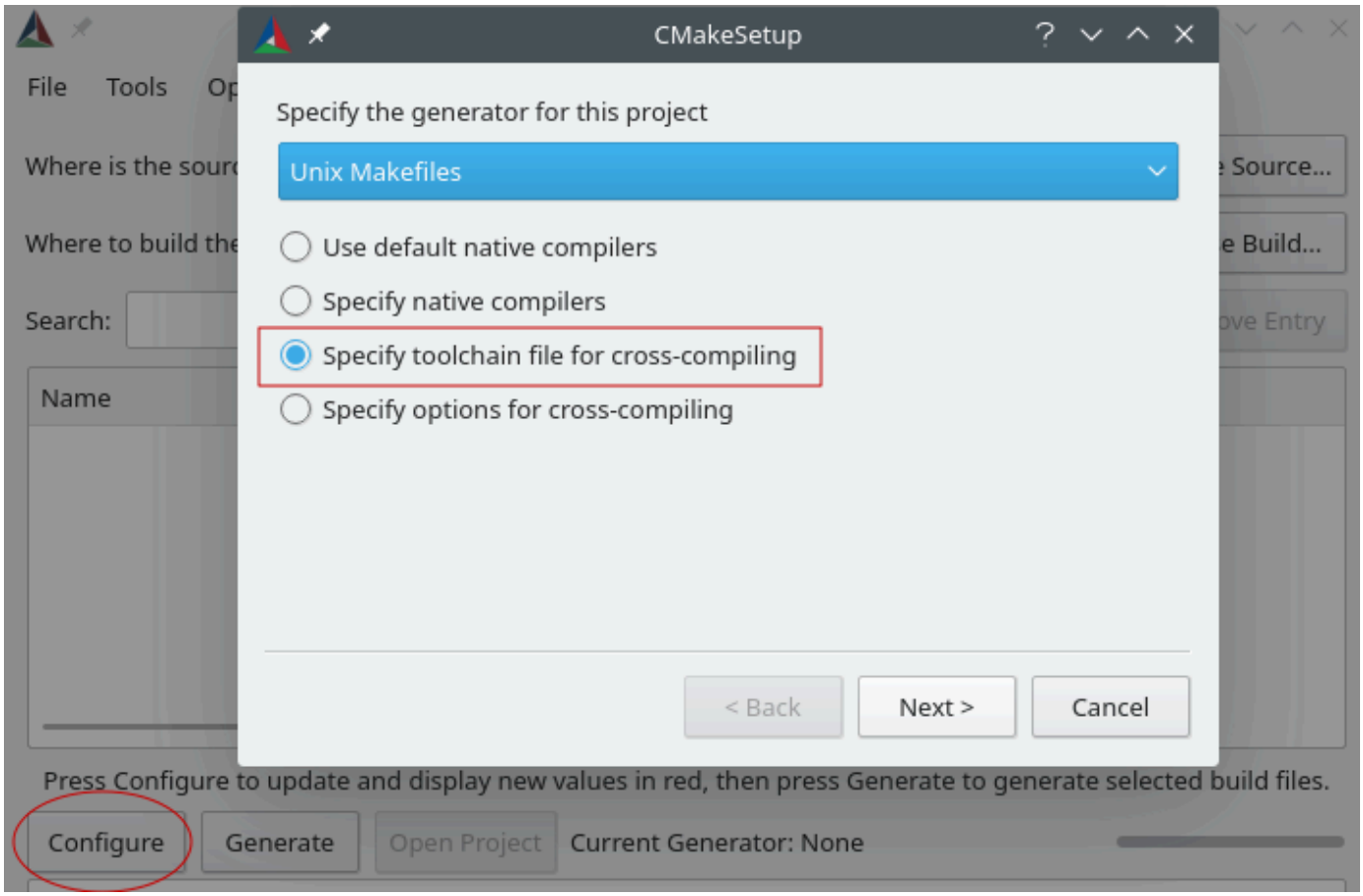
CMake GUI を使用して FreeRTOS ビルドファイルを生成できます。

CMake GUI でビルドファイルを生成するには

1. コマンドラインから cmake-gui を発行して GUI を起動します。
2. [Browse Source] (ソースの参照) を選択してソース入力を指定してから、[Browse Build] (ビルドの参照) を選択してビルド出力を指定します。



3. [Configure] (設定) を選択し、[Specify the build generator for this project] (このプロジェクトのビルドジェネレーターを指定) で、生成したビルドファイルを構築するために使用するビルドシステムを見つけて選択します。ポップアップウィンドウが表示されない場合は、既存のビルドディレクトリを再利用している可能性があります。この場合、[File] (ファイル) メニューの [Delete Cache] (キャッシュの削除) を選択して CMake キャッシュを削除します。



4. [Specify toolchain file for cross-compiling] (クロスコンパイル用にツールチェーンファイルを指定) を選択してから、[Next] (次へ) を選択します。

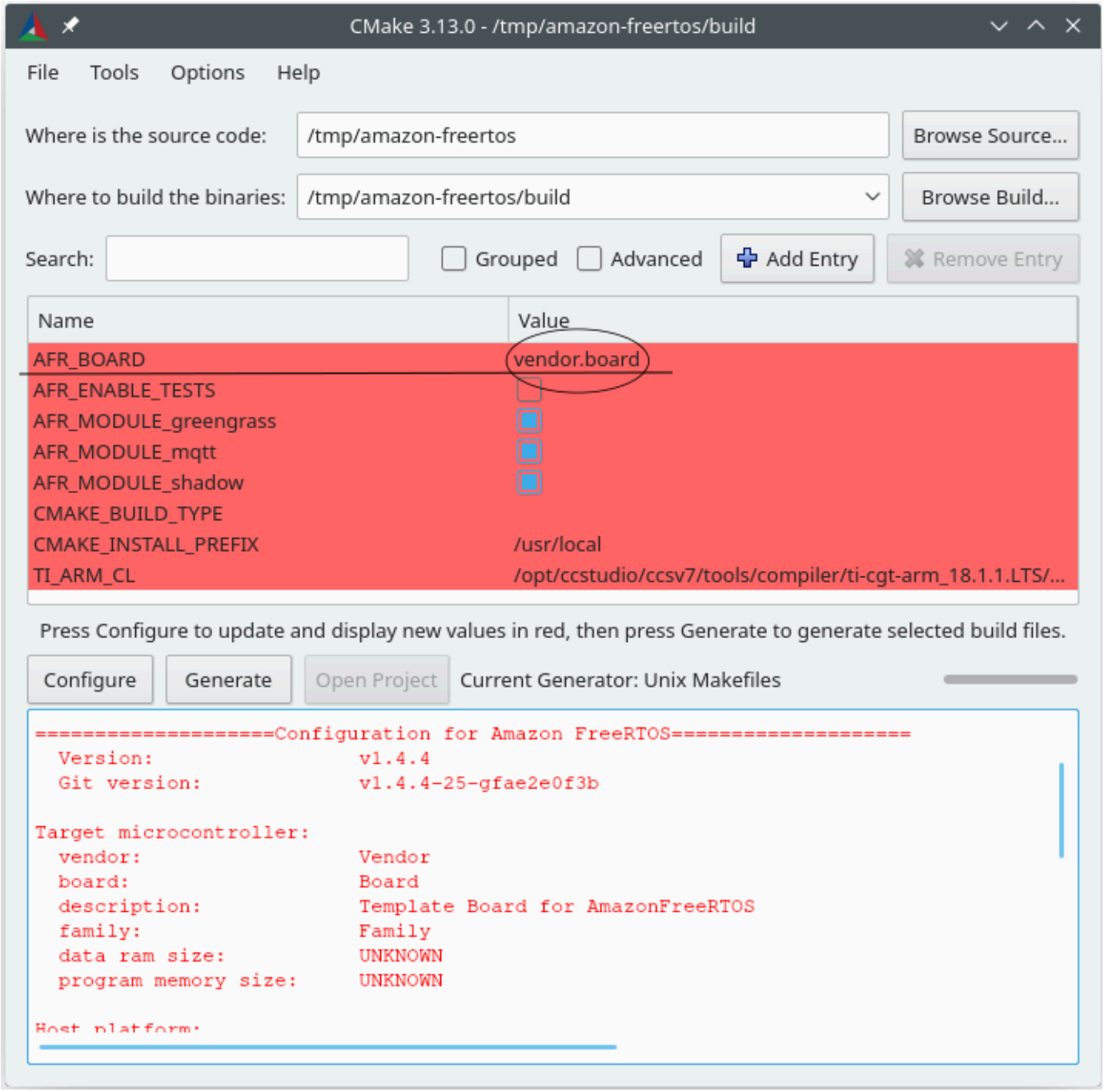
5. ツールチェーンファイル (*freertos*/tools/cmake/toolchains/arm-ti.cmake など) を選択し、[Finish] (完了) を選択します。

   FreeRTOS のデフォルト設定は、ポータブルレイヤーターゲットが提供されないテンプレートボードです。その結果、ウィンドウに Error in configuration process というメッセージが表示されます。

   > **Note**
   >
   > 次のエラーが表示される場合。
   >
   > ```
   > CMake Error at tools/cmake/toolchains/find_compiler.cmake:23 (message):
   > Compiler not found, you can specify search path with AFR_TOOLCHAIN_PATH.
   > ```

   コンパイラが PATH 環境変数にないことを示しています。GUI で AFR_TOOLCHAIN_PATH 変数を設定して、コンパイラをインストールした場所を CMake に伝えることができます。AFR_TOOLCHAIN_PATH 変数が見つからない場合、[Add Entry] (エントリの追加) を選択します。ポップアップウィンドウで、[Name] (名前) に **AFR_TOOLCHAIN_PATH** と入力します。[Compiler Path] (コンパイラパス) にコンパイラへのパスを入力します (C:/toolchains/arm-none-eabi-gcc など)。

6. GUI は次のようになります。



[AFR_BOARD] を選択し、ボードを選択してから、もう一度 [Configure] (設定) を選択します。

7. [Generate] (生成) を選択します。CMake はビルドシステムファイル (makefiles や ninja ファイルなど) を生成し、これらのファイルは最初のステップで指定したビルドディレクトリに表示されます。次のセクションの手順に従って、バイナリイメージを生成します。

生成されたビルドファイルから FreeRTOS を構築する

ネイティブビルドシステムによるビルド

出力バイナリディレクトリからビルドシステムコマンドを呼び出すことで、ネイティブビルドシステムで FreeRTOS を構築できます。

たとえば、ビルドファイルの出力ディレクトリが <build_dir> で、ネイティブビルドシステムとして Make を使用している場合は、次のコマンドを実行します。

```
cd <build_dir>
make -j4
```

CMake による の構築

CMake コマンドラインツールを使って FreeRTOS を構築することもできます。CMake にはネイティブビルドシステムを呼び出すための抽象化レイヤーが用意されています。例:

```
cmake --build build_dir
```

CMake コマンドラインツールのビルドモードのその他の一般的な用途は次のとおりです。

```
# 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
```

CMake ビルドモードの詳細については、「CMake ドキュメント」を参照してください。

開発者モードのキーのプロビジョニング

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-

FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

このセクションでは、信頼された X.509 クライアント証明書をラボテスト用に IoT デバイスに追加するための 2 つのオプションについて説明します。デバイスの機能に応じて、オンボード ECDSA キーの生成、プライベートキーのインポート、X.509 証明書の登録など、プロビジョニング関連のさまざまな操作がサポートされるかどうかが決まります。また、ユースケースに応じて、オンボードフラッシュストレージの使用から専用の暗号ハードウェアの使用まで、さまざまなレベルのキー保護が必要です。このセクションでは、デバイスの暗号化機能での作業ロジックを示します。

オプション #1: からのプライベートキーのインポート AWS IoT

ラボテストの目的で、デバイスでプライベートキーのインポートが許可されている場合は、「FreeRTOS デモを設定する」の手順に従います。

オプション #2: オンボードプライベートキーの生成

デバイスにセキュアエレメントがある場合、または独自のデバイスキーペアと証明書を生成する場合は、こちらの手順に従います。

初期設定

まず、「」のステップを実行しますがFreeRTOS デモを設定する、最後のステップはスキップします (つまり、 AWS IoT 認証情報をフォーマットするには を実行しないでください)。最終的に、demos/include/aws_clientcredential.h ファイルはお客様の設定で更新されますが、demos/include/aws_clientcredential_keys.h ファイルは更新されません。

デモプロジェクトの設定

ボード固有の入門ガイド のボードのガイドの説明に従って、coreMQTT Mutual Authentication デモを開きます。プロジェクトで、ファイル aws_dev_mode_key_provisioning.c を開き、デフォルトでゼロに設定されている keyprovisioningFORCE_GENERATE_NEW_KEY_PAIR の定義を 1 に変更します。

```
#define keyprovisioningFORCE_GENERATE_NEW_KEY_PAIR 1
```

その後、デモプロジェクトをビルドして実行し、次のステップに進みます。

パブリックキーの抽出

デバイスではまだプライベートキーとクライアント証明書がプロビジョニングされていないため、デモは AWS IoTに対する認証に失敗します。ただし、coreMQTT Mutual Authentication デモは、開発者モードのキープロビジョニングを実行することから始まり、プライベートキーがまだ存在しない場合は作成されます。シリアルコンソール出力の先頭近くは以下のようになっています。

```
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
```

キーバイトの 6 行を DevicePublicKeyAsciiHex.txt というファイルにコピーします。次に、コマンドラインツール「xxd」を使用して、16 進バイトをバイナリに解析します。

```
xxd -r -ps DevicePublicKeyAsciiHex.txt DevicePublicKeyDer.bin
```

「openssl」を使用して、バイナリエンコード (DER) デバイスのパブリックキーを PEM 形式に設定します。

```
openssl ec -inform der -in DevicePublicKeyDer.bin -pubin -pubout -outform pem -out
 DevicePublicKey.pem
```

上記で有効にした一時的なキー生成設定は必ず無効にしてください。無効にしない場合、デバイスでは別のキーペアが作成されるため、前の手順を繰り返す必要があります。

```
#define keyprovisioningFORCE_GENERATE_NEW_KEY_PAIR 0
```

PKI (公開鍵基盤) のセットアップ

「CA 証明書の登録」の手順に従って、デバイスラボ証明書の証明書階層を作成します。「CA 証明書を使用したデバイス証明書の作成」セクションで説明されているシーケンスを実行する前に停止します。

この場合、デバイスは証明書リクエスト (証明書サービスリクエスト (CSR)) に署名しません。CSR の作成と署名に必要な X.509 エンコードロジックは、ROM サイズを小さくするため

に、FreeRTOS デモプロジェクトから除外されているためです。代わりに、ラボテストの目的で、ワークステーションでプライベートキーを作成し、そのキーを使用して CSR に署名します。

```
openssl genrsa -out tempCsrSigner.key 2048
openssl req -new -key tempCsrSigner.key -out deviceCert.csr
```

認証機関を作成して に登録したら AWS IoT、次のコマンドを使用して、前のステップで署名されたデバイス CSR に基づいてクライアント証明書を発行します。

```
openssl x509 -req -in deviceCert.csr -CA rootCA.pem -CAkey rootCA.key
 -CAcreateserial -out deviceCert.pem -days 500 -sha256 -force_pubkey
 DevicePublicKey.pem
```

CSR は一時的なプライベートキーで署名されましたが、発行された証明書は実際のデバイスのプライベートキーでのみ使用できます。CSR 署名者キーを別のハードウェアに保存し、その特定のキーで署名されたリクエストに対してのみ証明書を発行するように認証機関を設定する場合は、その同じメカニズムを本番稼働に使用できます。そのキーは、指定された管理者の管理下にあることが必要です。

証明書のインポート

証明書が発行されたら、次のステップはその証明書をデバイスにインポートすることです。また、JITP の使用時に への初回認証 AWS IoT を成功させるために必要になるため、認証機関 (CA) 証明書をインポートする必要があります。プロジェクトの `aws_clientcredential_keys.h` ファイルで、`keyCLIENT_CERTIFICATE_PEM` マクロを deviceCert.pem の内容になるように設定し、`keyJITR_DEVICE_CERTIFICATE_AUTHORITY_PEM` マクロを `rootCA.pem` の内容になるように設定します。

デバイスの承認

「独自の証明書を使用する」の説明に従って、 を AWS IoT レジストリ`deviceCert.pem`にインポートします。新しい AWS IoT モノを作成し、保留中の証明書とポリシーをモノにアタッチしてから、証明書を ACTIVE としてマークする必要があります。これらのステップはすべて、AWS IoT コンソールで手動で実行できます。

新しいクライアント証明書が ACTIVE になり、モノとポリシーに関連付けられたら、coreMQTT Mutual Authentication デモをもう一度実行します。今回は、 AWS IoT MQTT ブローカーへの接続が成功します。

ボード固有の入門ガイド

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

最初のステップ を完成した後、FreeRTOS の使用開始に関するボード固有の手順については、ボードのガイドを参照してください。

- Cypress CYW943907AEVAL1F 開発キットの開始方法
- Cypress CYW954907AEVAL1F 開発キットの開始方法
- Cypress CY8CKIT-064S0S2-4343W キットの開始方法
- Infineon XMC4800 IoT 接続キットの開始方法
- MW32x AWS IoT スターターキットの開始方法
- MediaTek MT7697Hx Development Kit の開始方法
- Microchip Curiosity PIC32MZ EF の開始方法
- Nuvoton NuMaker-IoT-M487 の開始方法
- NXP LPC54018 IoT モジュールの開始方法
- Renesas Starter Kit+ for RX65N-2MB の開始方法
- STMicroelectronics STM32L4 ディスカバリキット IoT ノード用の開始方法
- Texas Instruments CC3220SF-LAUNCHXL の開始方法
- Windows Device Simulator の開始方法
- Xilinx Avnet MicroZed Industrial IoT キットの開始方法

> **Note**
>
> FreeRTOS の以下の自己完結型入門ガイドの 最初のステップ を完了する必要はありません。
>
> - Windows Simulator で Microchip ATECC608A セキュアエレメントの使用の開始方法

- Espressif ESP32-DevKitC と ESP-WROVER-KIT の開始方法
- Espressif ESP32-WROOM-32SE の開始方法
- Espressif ESP32-S2 の開始方法
- Infineon OPTIGA Trust X と XMC4800 IoT Connectivity Kit の開始方法
- Nordic nRF52840-DK の開始方法

Cypress CYW943907AEVAL1F 開発キットの開始方法

Important

このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Cypress CYW943907AEVAL1F 開発キットの使用を開始するための手順について説明します。Cypress CYW943907AEVAL1F 開発キットがない場合は、 AWS Partner Device Catalog にアクセスして当社のいずれかのパートナーから購入してください。

Note

このチュートリアルでは、coreMQTT Mutual Authentication デモをセットアップして実行する手順を説明します。このボードの FreeRTOS ポートは現在 TCP サーバーとクライアントのデモをサポートしていません。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。

Important

- このトピックでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

- *freertos* パスにスペース文字が含まれていると、構築が失敗する可能性があります。リポジトリをクローンまたはコピーするときは、作成するパスにスペース文字が含まれていないことを確認してください。
- Microsoft Windows でのファイルパスの最大長は 260 文字です。FreeRTOS のダウンロードディレクトリパスが長くなると、構築が失敗する可能性があります。
- ソースコードにはシンボリックリンクが含まれている可能性があるため、Windows を使用してアーカイブを抽出する場合は、次の操作を行う必要があります。
  - 開発者モードを有効にするか、または、
  - 管理者としてコンソールを使用します。

  この操作を行えば、Windows でアーカイブを抽出する際にシンボリックリンクを適切に作成できます。この操作を行わないと、シンボリックリンクは、そのパスがテキストとして含まれる、または空白の通常ファイルとして書き込まれます。詳細については、ブログの投稿「Symlinks in Windows 10!」を参照してください。

  Windows で Git を使用する場合は、開発者モードを有効にするか、以下を実行する必要があります。
  - 次のコマンドを使用して、`core.symlinks` を true に設定します。

    ```
    git config --global core.symlinks true
    ```

  - システムへの書き込みを行う git コマンド (git pull、git clone、git submodule update --init --recursive など) を使用する場合は、管理者としてコンソールを使用します。
- FreeRTOS をダウンロードする で説明したように、Cypress の FreeRTOS 移植は、現在 GitHub でのみ利用できます。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

4. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

開発環境のセットアップ

WICED Studio SDK をダウンロードおよびインストールする

この入門ガイドでは、Cypress WICED Studio SDK を使用して FreeRTOS デモでボードをプログラミングします。WICED ソフトウェアウェブサイトにアクセスし、Cypress から WICED Studio SDK をダウンロードします。このソフトウェアをダウンロードするには、無料の Cypress アカウントに登録する必要があります。WICED Studio SDK は、Windows、macOS、および Linux オペレーティングシステムと互換性があります。

> **Note**
>
> オペレーティングシステムによっては、追加のインストール手順が必要になります。インストールする WICED Studio のオペレーティングシステムとバージョンのインストール手順をすべてお読みの上、記載されている手順に従ってください。

環境変数を設定する

WICED Studio を使用してボードをプログラムする前に、WICED Studio SDK インストールディレクトリ用の環境変数を作成する必要があります。変数の作成中に WICED Studio が実行されている場合は、変数を設定した後にアプリケーションを再起動する必要があります。

> **Note**
>
> WICED Studio インストーラーは、マシン上に `WICED-Studio-`*`m`*`.`*`n`* という名前が付けられた 2 つの個別のフォルダーを作成します。ここで `m` および `n` は、それぞれメジャーバージョン番号とマイナーバージョン番号です。このドキュメントでは、フォルダー名 `WICED-Studio-6.2` を想定していますが、インストールするバージョンの正しい名前を使用してください。`WICED_STUDIO_SDK_PATH` 環境変数を定義する場合は、WICED Studio IDE のインストールパスではなく、必ず WICED Studio SDK の完全なインストールパスを指定してください。Windows および macOS では、SDK の `WICED-Studio-`*`m`*`.`*`n`* フォルダーは、デフォルトで `Documents` フォルダーに作成されます。

Windows で環境変数を作成するには

1. [Control Panel] (コントロールパネル) を開き、[System] (システム)、[Advanced System Settings] (システムの詳細設定) の順に選択します。
2. [Advanced] (詳細設定) タブで、[Environment Variables] (環境変数) を選択します。
3. [User variables] (ユーザー変数) で 、[New] (新規) を選択します。
4. [Variable name] (変数名) で、**`WICED_STUDIO_SDK_PATH`** と入力します。[Variable value] (変数値) に、WICED Studio SDK のインストールディレクトリを入力します。

Linux または macOS で環境変数を作成するには

1. マシンで `/etc/profile` ファイルを開き、以下の値をファイルの最終行に追加します。

   ```
   export WICED_STUDIO_SDK_PATH=installation-path/WICED-Studio-6.2
   ```

2. マシンを再起動します。
3. ターミナルを開き、次のコマンドを実行します。

   ```
   cd freertos/vendors/cypress/WICED_SDK
   ```

   ```
   perl platform_adjust_make.pl
   ```

   ```
   chmod +x make
   ```

シリアル接続の確立

ホストマシンとボードの間にシリアル接続を確立するには

1. USB 標準 A - Micro-B ケーブルを使用して、ボードをホストコンピュータに接続します。
2. ホストコンピュータのボードに接続するための USB シリアルポート番号を確認します。
3. シリアルターミナルを起動し、以下の設定で接続を開きます。

   - ボーレート: 115200
   - データ: 8 ビット
   - パリティ: なし

- ストップビット: 1
- フロー制御: なし

ターミナルのインストールとシリアル接続の設定に関する詳細については、「ターミナルエミュレーターをインストールする」を参照してください。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

FreeRTOS デモプロジェクトを構築して実行する

ボードにシリアル接続を設定したら、FreeRTOS デモプロジェクトを構築し、ボードにデモをフラッシュしてからデモを実行できます。

WICED Studio で FreeRTOS デモプロジェクトを構築して実行するには

1. WICED Studio を起動します。
2. [File] (ファイル) メニューから [Import] (インポート) を選択します。[General] フォルダーを展開し、[Existing Projects into Workspace] (既存のプロジェクトを WorkSpace へ) を選択してから、[Next] (次へ) を選択します。
3. [Select root directory] (ルートディレクトリの選択) で、[Browse...] (参照) を選択し、パス *freertos*/projects/cypress/CYW943907AEVAL1F/wicedstudio に移動して、[OK] を選択します。
4. [Projects] (プロジェクト) で、[aws_demo] プロジェクトのみのチェックボックスをオンにします。[Finish] (完了) を選択してプロジェクトをインポートします。ターゲットプロジェクト aws_demo が [Make Target] (ターゲットの作成) ウィンドウに表示されます。

5. [WICED Platform] (WICED プラットフォーム) メニューを展開し、[WICED Filters off] (WICED フィルターオフ) を選択します。
6. [Make Target] (ターゲットの作成) ウィンドウで [aws_demo] を展開して、demo.aws_demo ファイルを右クリックし、[Build Target] (ターゲットを構築) を選択してデモを構築してからボードにダウンロードします。デモは、構築されボードにダウンロードされた後、自動的に実行されます。

トラブルシューティング

- Windows を使用している場合は、デモプロジェクトのビルドおよび実行時に以下のエラーが表示される場合があります。

```
: recipe for target 'download_dct' failed
make.exe[1]: *** [download_dct] Error 1
```

  このエラーを解決するには、以下のように行います。

  1. *WICED-Studio-SDK-PATH*\WICED-Studio-6.2\43xxx_Wi-Fi\tools\OpenOCD\Win32 を参照して、openocd-all-brcm-libftdi.exe をダブルクリックします。
  2. *WICED-Studio-SDK-PATH*\WICED-Studio-6.2\43xxx_Wi-Fi\tools\drivers\CYW9WCD1EVAL1 を参照して、InstallDriver.exe をダブルクリックします。

- Linux または macOS を使用している場合は、デモプロジェクトのビルドおよび実行時に以下のエラーが表示される場合があります。

```
make[1]: *** [download_dct] Error 127
```

  このエラーをトラブルシューティングするには、以下のコマンドを使用して、libusb-dev パッケージを更新します。

```
sudo apt-get install libusb-dev
```

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Cypress CYW954907AEVAL1F 開発キットの開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Cypress CYW954907AEVAL1F 開発キットの使用を開始するための手順について説明します。Cypress CYW954907AEVAL1F 開発キットがない場合は、 AWS Partner Device Catalog にアクセスして当社のいずれかのパートナーから購入してください。

> **Note**
>
> このチュートリアルでは、coreMQTT Mutual Authentication デモをセットアップして実行する手順を説明します。このボードの FreeRTOS 移植は現在 TCP サーバーとクライアントのデモをサポートしていません。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。

> **Important**
>
> - このトピックでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。
> - *`freertos`* パスにスペース文字が含まれていると、構築が失敗する可能性があります。リポジトリをクローンまたはコピーするときは、作成するパスにスペース文字が含まれていないことを確認してください。
> - Microsoft Windows でのファイルパスの最大長は 260 文字です。FreeRTOS のダウンロードディレクトリパスが長くなると、構築が失敗する可能性があります。
> - ソースコードにはシンボリックリンクが含まれている可能性があるため、Windows を使用してアーカイブを抽出する場合は、次の操作を行う必要があります。
>   - 開発者モードを有効にするか、または、

- 管理者としてコンソールを使用します。

この操作を行えば、Windows でアーカイブを抽出する際にシンボリックリンクを適切に作成できます。この操作を行わないと、シンボリックリンクは、そのパスがテキストとして含まれる、または空白の通常ファイルとして書き込まれます。詳細については、ブログの投稿「Symlinks in Windows 10!」を参照してください。

Windows で Git を使用する場合は、開発者モードを有効にするか、以下を実行する必要があります。

- 次のコマンドを使用して、`core.symlinks` を true に設定します。

```
git config --global core.symlinks true
```

- システムへの書き込みを行う git コマンド (git pull、git clone、git submodule update --init --recursive など) を使用する場合は、管理者としてコンソールを使用します。

- FreeRTOS をダウンロードする で説明したように、Cypress の FreeRTOS 移植は、現在 GitHub でのみ利用できます。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
4. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

開発環境のセットアップ

WICED Studio SDK をダウンロードおよびインストールする

この入門ガイドでは、Cypress WICED Studio SDK を使用して FreeRTOS デモでボードをプログラミングします。WICED ソフトウェアウェブサイトにアクセスし、Cypress から WICED Studio SDK をダウンロードします。このソフトウェアをダウンロードするには、無料の Cypress アカウントに

登録する必要があります。WICED Studio SDK は、Windows、macOS、および Linux オペレーティングシステムと互換性があります。

> **Note**
>
> オペレーティングシステムによっては、追加のインストール手順が必要になります。インストールする WICED Studio のオペレーティングシステムとバージョンのインストール手順をすべてお読みの上、記載されている手順に従ってください。

環境変数を設定する

WICED Studio を使用してボードをプログラムする前に、WICED Studio SDK インストールディレクトリ用の環境変数を作成する必要があります。変数の作成中に WICED Studio が実行されている場合は、変数を設定した後にアプリケーションを再起動する必要があります。

> **Note**
>
> WICED Studio インストーラーは、マシン上に `WICED-Studio-m.n` という名前が付けられた 2 つの個別のフォルダーを作成します。ここで `m` および `n` は、それぞれメジャーバージョン番号とマイナーバージョン番号です。このドキュメントでは、フォルダー名 `WICED-Studio-6.2` を想定していますが、インストールするバージョンの正しい名前を使用してください。`WICED_STUDIO_SDK_PATH` 環境変数を定義する場合は、WICED Studio IDE のインストールパスではなく、必ず WICED Studio SDK の完全なインストールパスを指定してください。Windows および macOS では、SDK の `WICED-Studio-m.n` フォルダーは、デフォルトで `Documents` フォルダーに作成されます。

Windows で環境変数を作成するには

1. [Control Panel] (コントロールパネル) を開き、[System] (システム)、[Advanced System Settings] (システムの詳細設定) の順に選択します。
2. [Advanced] (詳細設定) タブで、[Environment Variables] (環境変数) を選択します。
3. [User variables] (ユーザー変数) で 、[New] (新規) を選択します。
4. [Variable name] (変数名) で、**`WICED_STUDIO_SDK_PATH`** と入力します。[Variable value] (変数値) に、WICED Studio SDK のインストールディレクトリを入力します。

Linux または macOS で環境変数を作成するには

1. マシンで /etc/profile ファイルを開き、以下の値をファイルの最終行に追加します。

```
export WICED_STUDIO_SDK_PATH=installation-path/WICED-Studio-6.2
```

2. マシンを再起動します。
3. ターミナルを開き、次のコマンドを実行します。

```
cd freertos/vendors/cypress/WICED_SDK
```

```
perl platform_adjust_make.pl
```

```
chmod +x make
```

シリアル接続の確立

ホストマシンとボードの間にシリアル接続を確立するには

1. USB 標準 A - Micro-B ケーブルを使用して、ボードをホストコンピュータに接続します。
2. ホストコンピュータのボードに接続するための USB シリアルポート番号を確認します。
3. シリアルターミナルを起動し、以下の設定で接続を開きます。

   - ボーレート: 115200
   - データ: 8 ビット
   - パリティ: なし
   - ストップビット: 1
   - フロー制御: なし

ターミナルのインストールとシリアル接続の設定に関する詳細については、「ターミナルエミュレーターをインストールする」を参照してください。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

FreeRTOS デモプロジェクトを構築して実行する

ボードにシリアル接続を設定したら、FreeRTOS デモプロジェクトを構築し、ボードにデモをフラッシュしてからデモを実行できます。

WICED Studio で FreeRTOS デモプロジェクトを構築して実行するには

1. WICED Studio を起動します。
2. [File] (ファイル) メニューから [Import] (インポート) を選択します。[General] フォルダーを展開し、[Existing Projects into Workspace] (既存のプロジェクトを WorkSpace へ) を選択してから、[Next] (次へ) を選択します。
3. [Select root directory] (ルートディレクトリの選択) で、[Browse...] (参照) を選択し、パス *freertos*/projects/cypress/CYW954907AEVAL1F/wicedstudio に移動して、[OK] を選択します。
4. [Projects] (プロジェクト) で、[aws_demo] プロジェクトのみのチェックボックスをオンにします。[Finish] (完了) を選択してプロジェクトをインポートします。ターゲットプロジェクト aws_demo が [Make Target] (ターゲットの作成) ウィンドウに表示されます。
5. [WICED Platform] (WICED プラットフォーム) メニューを展開し、[WICED Filters off] (WICED フィルターオフ) を選択します。
6. [Make Target] (ターゲットの作成) ウィンドウで [aws_demo] を展開して、demo.aws_demo ファイルを右クリックし、[Build Target] (ターゲットを構築) を選択してデモを構築してからボードにダウンロードします。デモは、構築されボードにダウンロードされた後、自動的に実行されます。

トラブルシューティング

- Windows を使用している場合は、デモプロジェクトのビルドおよび実行時に以下のエラーが表示される場合があります。

```
: recipe for target 'download_dct' failed
make.exe[1]: *** [download_dct] Error 1
```

このエラーを解決するには、以下のように行います。

1. *WICED-Studio-SDK-PATH*\WICED-Studio-6.2\43xxx_Wi-Fi\tools\OpenOCD\Win32 を参照して、openocd-all-brcm-libftdi.exe をダブルクリックします。
2. *WICED-Studio-SDK-PATH*\WICED-Studio-6.2\43xxx_Wi-Fi\tools\drivers\CYW9WCD1EVAL1 を参照して、InstallDriver.exe をダブルクリックします。

- Linux または macOS を使用している場合は、デモプロジェクトのビルドおよび実行時に以下のエラーが表示される場合があります。

```
make[1]: *** [download_dct] Error 127
```

このエラーをトラブルシューティングするには、以下のコマンドを使用して、libusb-dev パッケージを更新します。

```
sudo apt-get install libusb-dev
```

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Cypress CY8CKIT-064S0S2-4343W キットの開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、CY8CKIT-064S0S2-4343W キットの使用を開始するための手順について説明します。お持ちでない場合は、リンクを使用してキットを購入できます。このリンクを使用して、キットのユーザーガイドにアクセスすることもできます。

入門

開始する前に、デバイスを AWS クラウドに接続するように AWS IoT と FreeRTOS を設定する必要があります。手順については、最初のステップ を参照してください。前提条件を完了すると、 AWS IoT Core 認証情報を含む FreeRTOS パッケージが作成されます。

> **Note**
>
> このチュートリアルでは、「最初のステップ」セクションで作成した FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

開発環境のセットアップ

FreeRTOS では CMake または Make 構築フローのいずれかを使用できます。Make 構築フローには、ModusToolbox を使用できます。ModusToolbox に付属している Eclipse IDE か、IAR EW-Arm、Arm MDK、Microsoft Visual Studio Code などのパートナー IDE を使用できます。Eclipse IDE は、Windows、macOS、および Linux オペレーティングシステムと互換性があります。

始める前に、最新の ModusToolbox ソフトウェアをダウンロードして、インストールします。詳細については、ModusToolbox インストールガイドを参照してください。

ModusToolbox 2.1 以前のツールを更新する

ModusToolbox 2.1 Eclipse IDE を使用してこのキットをプログラムする場合は、OpenOCD および Firmware-loader ツールを更新する必要があります。

以下のステップでは、デフォルトの *ModusToolbox* パスは次のようになります。

- Windows では `C:\Users\`*user_name*`\ModusToolbox` です。
- Linux では *user_home*`/ModusToolbox`、またはアーカイブファイルを抽出する場所になります。
- MacOS では、ウィザードで選択したボリュームの [Applications] (アプリケーション) フォルダの下です。

OpenOCD の更新

このキットでチップを正常に消去してプログラムするには、Cypress OpenOCD 4.0.0 以降が必要です。

Cypress OpenOCD を更新するには

1. Cypress OpenOCD リリースページにアクセスします。
2. OS (Windows/Mac/Linux) 用のアーカイブファイルをダウンロードします。
3. *ModusToolbox*/tools_2.x/openocd にある既存のファイルを削除します。
4. *ModusToolbox*/tools_2.x/openocd にあるファイルを前のステップでダウンロードしたアーカイブの抽出ファイルで置き換えます。

Firmware-loader の更新

このキットには、Cypress Firmware-loader 3.0.0 以降が必要です。

Cypress Firmware-loader を更新するには

1. Cypress Firmware-loader リリースページにアクセスします。
2. OS (Windows/Mac/Linux) 用のアーカイブファイルをダウンロードします。
3. *ModusToolbox*/tools_2.x/fw-loader にある既存のファイルを削除します。
4. *ModusToolbox*/tools_2.x/fw-loader にあるファイルを前のステップでダウンロードしたアーカイブの抽出ファイルで置き換えます。

または、CMake を使用して FreeRTOS アプリケーションのソースコードからプロジェクトビルドファイルを生成し、好みのビルドツールを使用してプロジェクトを構築してから、OpenOCD を使用してキットをプログラムすることもできます。CMake フローでプログラミングに GUI ツールを使用する場合は、Cypress Programmer を Cypress プログラミングソリューションウェブページからダウンロードして、インストールします。詳細については、「FreeRTOS で CMake を使用する」を参照してください。

ハードウェアの設定

キットのハードウェアをセットアップするには、次の手順を実行します。

1. キットをプロビジョニングする

   CY8CKIT-064S0S2-4343W キットのプロビジョニングガイドの手順に従って、 AWS IoT用にキットを安全にプロビジョニングします。

   このキットには、CySecureTools 3.1.0 以降が必要です。
2. シリアル接続を設定する

a. キットをホストコンピュータに接続します。

b. キットの USB シリアルポートは、ホストコンピュータに自動的に表示されます。ポート番号を指定します。Windows では、デバイスマネージャーの [Ports] (ポート) (COM と LPT) を使用して指定できます。

c. シリアルターミナルを起動し、以下の設定で接続を開きます。

- ボーレート: 115200
- データ: 8 ビット
- パリティ: なし
- ストップビット: 1
- フロー制御: なし

FreeRTOS デモプロジェクトを構築して実行する

このセクションでは、デモを構築して実行します。

1. CY8CKIT-064S0S2-4343W キットのプロビジョニングガイドの手順に従います。
2. FreeRTOS デモを構築します。

a. ModusToolbox の Eclipse IDE を開き、ワークスペースを選択するか、または作成します。

b. [File] (ファイル) メニューから [Import] (インポート) を選択します。

[General] (全般) を展開し、[Existing Projects into Workspace] (既存のプロジェクトを WorkSpace へ) を選択してから、[Next] (次へ) を選択します。

c. [Root Directory] (ルートディレクトリ) に *freertos*/projects/cypress/CY8CKIT-064S0S2-4343W/mtb/aws_demos と入力し、プロジェクト名 aws_demos を選択します。デフォルトで、このプロジェクト名が選択されています。

d. [Finish] (完了) を選択してプロジェクトをワークスペースにインポートします。

e. 次のいずれかを実行して、アプリケーションを構築します。

- [Quick Panel] (クイックパネル) から、[Build aws_demos Application] (aws_demos アプリケーションの構築) を選択します。
- [Project] (プロジェクト) を選択して、[Build All] (すべて構築) を選択します。

プロジェクトのコンパイルにエラーがないことを確認します。

3. クラウド上の MQTT メッセージのモニタリング

デモを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには、次の手順に従います。

a. AWS IoT コンソール にサインインします。

b. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。

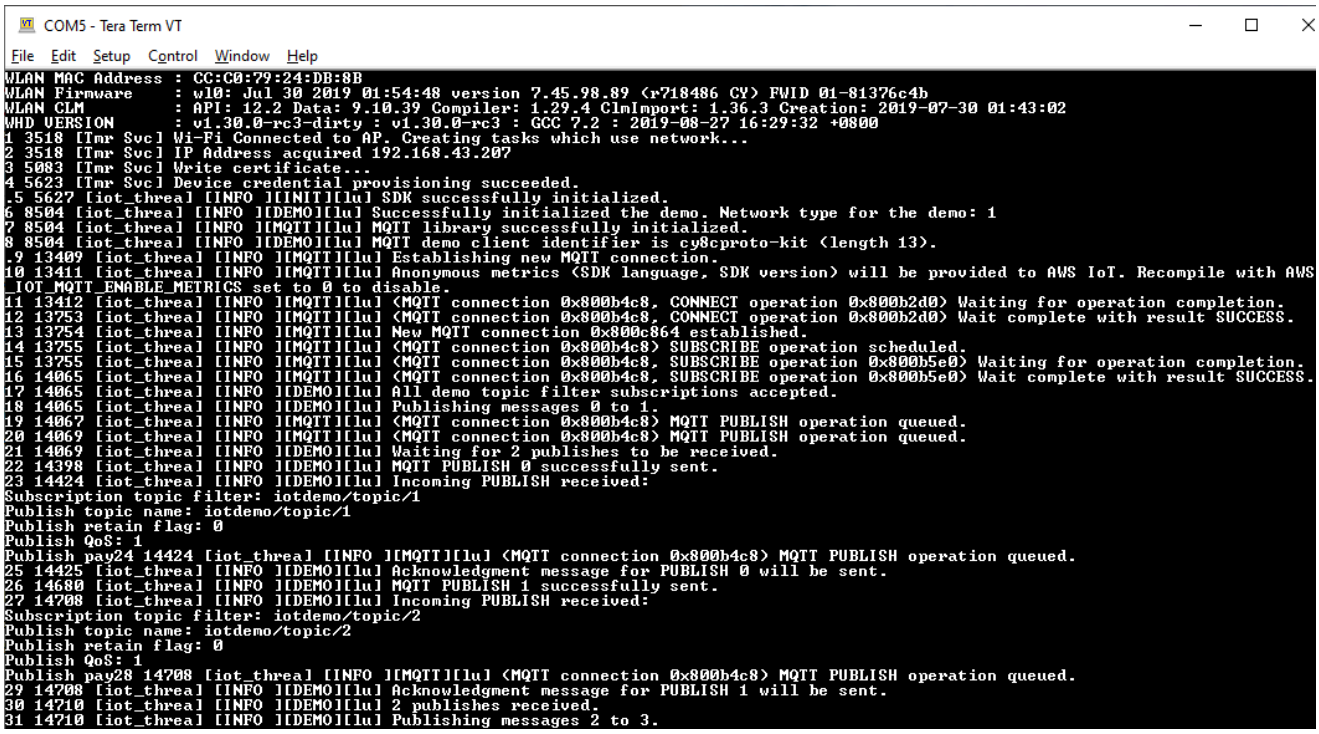
c. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

4. FreeRTOS デモプロジェクトを実行する

a. ワークスペースでプロジェクト `aws_demos` を選択します。

b. [Quick Panel] (クイックパネル) から、[aws_demos Program (KitProg3)] (aws_demos プログラム (KitProg3)) を選択します。これにより、ボードがプログラムされ、プログラミングが完了した後にデモアプリケーションの実行が開始されます。

c. シリアルターミナルで実行中のアプリケーションのステータスを表示できます。次の図は、ターミナル出力の一部を示しています。



MQTT デモは、4 つの異なるトピック (`iotdemo/topic/`*n*、n は 1~4) に関するメッセージを公開し、同じメッセージを受信するためにこれらのトピックをすべてサブスクライブ

します。メッセージが受信されると、デモはトピックに関する確認メッセージ iotdemo/acknowledgements を公開します。次のリストは、ターミナル出力に表示されるデバッグメッセージと、そのメッセージのシリアル番号を示しています。出力では、WICED Host Driver (WHD) ドライバーの詳細がシリアル番号なしで最初に表示されます。

1. 1 ~ 4 – デバイスは設定されたアクセスポイント (AP) に接続し、設定されたエンドポイントと証明書を使用して AWS サーバーに接続することでプロビジョニングされます。
2. 5 ~ 13: coreMQTT ライブラリが初期化され、デバイスが MQTT 接続を確立します。
3. 14 ~ 17: デバイスはすべてのトピックをサブスクライブして、公開されたメッセージを受信します。
4. 18 ~ 30: デバイスは 2 つのメッセージを公開し、それらを受信するまで待機します。各メッセージが受信されると、デバイスは確認メッセージを送信します。

すべてのメッセージが公開されるまで、公開、受信、および確認の同じサイクルが続きます。設定されたサイクル数が完了するまで、サイクルごとに 2 つのメッセージが公開されます。

5. FreeRTOS で CMake を使用する

CMake を使用してデモアプリケーションを構築して実行することもできます。CMake とネイティブビルドシステムをセットアップするには、「前提条件」を参照してください。

a. ビルドファイルを生成するには、次のコマンドを使用します。-DBOARD オプションを使用してターゲットボードを指定します。

```
cmake -DVENDOR=cypress -DBOARD=CY8CKIT_064S0S2_4343W -DCOMPILER=arm-gcc -
S freertos -B build_dir
```

Windows を使用する場合、CMake はデフォルトで Visual Studio を使用するため、-G オプションでネイティブビルドシステムを指定する必要があります。

Example

```
cmake -DVENDOR=cypress -DBOARD=CY8CKIT_064S0S2_4343W -DCOMPILER=arm-gcc -
S freertos -B build_dir -G Ninja
```

シェルパス内に arm-none-eabi-gcc がない場合は、AFR_TOOLCHAIN_PATH CMake 変数を設定する必要もあります。

Example

```
-DAFR_TOOLCHAIN_PATH=/home/user/opt/gcc-arm-none-eabi/bin
```

b. CMake を使用してプロジェクトを構築するには、次のコマンドを実行します。

```
cmake --build build_dir
```

c. 最後に、Cypress Programmer を使用して *build_dir* に生成された `cm0.hex` と `cm4.hex` ファイルをプログラムします。

その他のデモを実行する

次のデモアプリケーションは、現在のリリースで動作することがテストされ、確認されています。これらのデモは、*freertos*/demos ディレクトリにあります。デモの実行方法については、「FreeRTOS デモ」を参照してください。

- Bluetooth Low Energy デモ
- 無線通信経由更新デモ
- セキュアソケットエコークライアントのデモ
- AWS IoT Device Shadow デモ

デバッグ

キットの KitProg3 は SWD プロトコルでのデバッグをサポートしています。

- FreeRTOS アプリケーションをデバッグするには、ワークスペースで [aws_demos project] (aws_demos プロジェクト) を選択し、[Quick Panel] (クイックパネル) から [aws_demos Debug (KitProg3)] (aws_demos デバッグ (KitProg3)) を選択します。

OTA の更新

PSoC 64 MCU は、必要なすべての FreeRTOS 認定テストに合格しています。ただし、PSoC 64 Standard Secure AWS ファームウェアライブラリに実装されているオプションのover-the-air (OTA) 機能は、まだ評価中です。実装されている OTA 機能は、aws_ota_test_case_rollback_if_unable_to_connect_after_update.py を除き、現在 OTA 認定テストのすべてに合格しています。

正常に検証された OTA イメージが PSoC64 Standard Secure – AWS MCU を使用してデバイスに適用され、デバイスが通信できない場合 AWS IoT Core、デバイスは元の既知の正常なイメージに自動的にロールバックできません。これにより、デバイスが からアクセスできなくなり AWS IoT Core 、さらに更新できなくなる可能性があります。この機能は、Cypress チームによってまだ開発中です。

詳細については、「 による OTA 更新 AWS 」およびCY8CKIT-064S0S2-4343W キット」を参照してください。その他の質問がある場合、またはテクニカルサポートが必要な場合は、Cypress Developer Community にお問い合わせください。

Windows Simulator で Microchip ATECC608A セキュアエレメントの使用の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Windows Simulator で Microchip ATECC608A セキュアエレメントの使用を開始するための手順について説明します。

以下のハードウェアが必要です。

- Microchip ATECC608A セキュアエレメントクリックボード
- SAMD21 XPlained Pro
- mikroBUS Xplained Pro アダプター

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。

概要

このチュートリアルでは、以下の手順が含まれています。

1. ボードをホストマシンに接続します。

2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

Microchip ATECC608A ハードウェアのセットアップ

Microchip ATECC608A デバイスを操作する前に、まず SAMD21 をプログラムする必要があります。

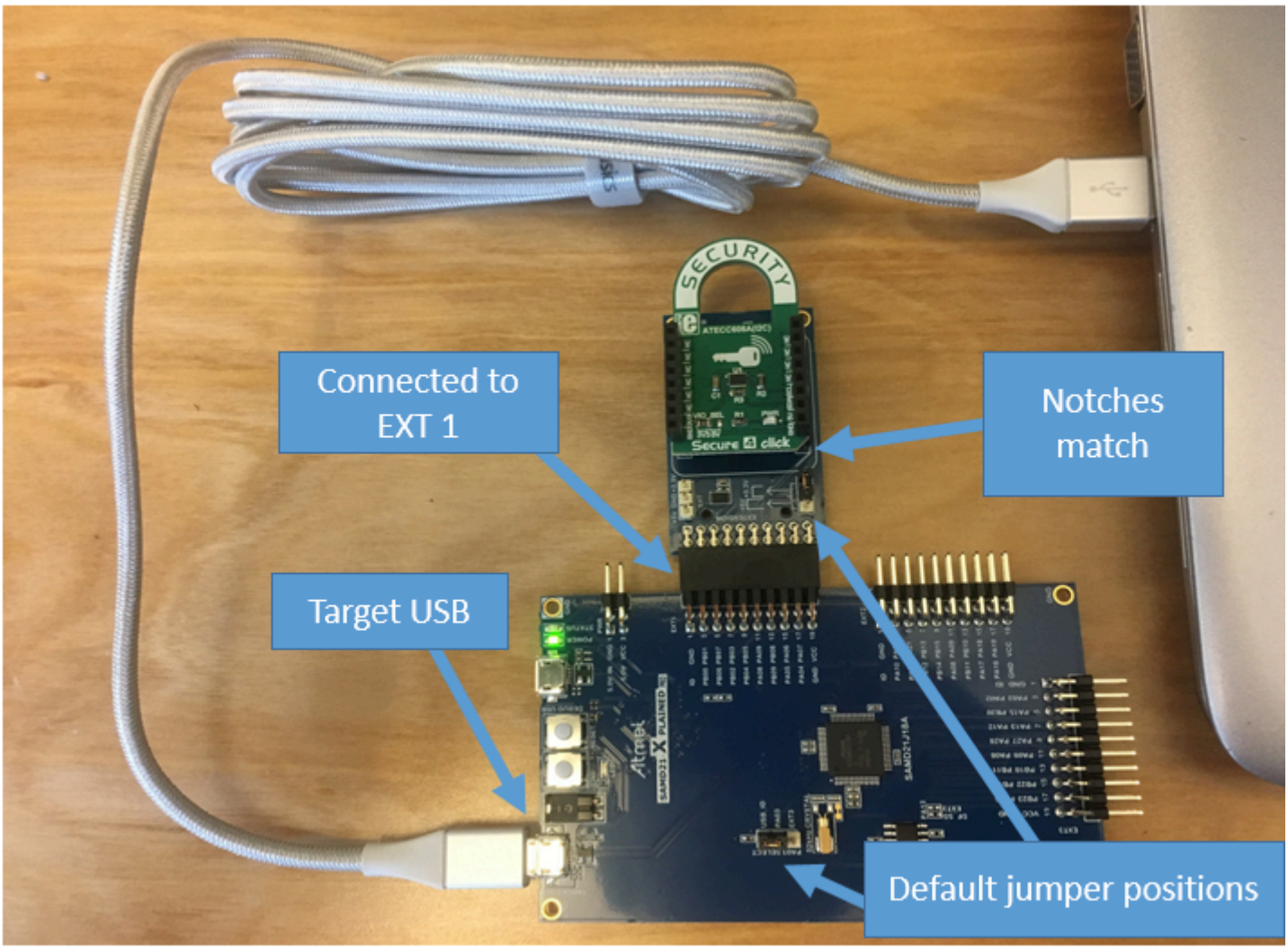
SAMD21 XPlained Pro ボードのセットアップ

1. CryptoAuthSSH-XSTK (DM320109) - Latest Firmware リンクをたどって、手順書 (PDF)、および D21 にプログラムできるバイナリを含む .zip ファイルをダウンロードします。
2. Atmel Studio 7 IDP をダウンロードしてインストールします。インストール時に必ず SMART ARM MCU ドライバアーキテクチャを選択するようにしてください。
3. USB 2.0 Micro B ケーブルを使用して「Debug USB」コネクタをコンピュータに接続し、PDF の指示に従います。(「Debug USB」コネクタは、POWER LED とピンに最も近い USB ポートです)。

ハードウェアを接続する

1. 「Debug USB」コネクタからマイクロ USB ケーブルを取り外します。
2. mikroBUS XPlained Pro アダプターを EXT1 位置の SAMD21 ボードに差し込みます。
3. ATECC608A セキュア 4 クリックボードを mikroBUSX XPlained Pro アダプターに差し込みます。クリックボードのノッチ付きコーナーがアダプタボードのノッチ付きアイコンと一致していることを確認します。
4. マイクロ USB ケーブルをターゲット USB に差し込みます。

セットアップは以下のようになります。



開発環境をセットアップする

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があ

ります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

  アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。
- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。
- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

設定

1. FreeRTOS GitHub リポジトリから FreeRTOS レポをダウンロードします。

   GitHub から FreeRTOS をダウンロードするには

   1. FreeRTOS GitHub リポジトリを参照します。
   2. [Clone or download] (クローンまたはダウンロード) を選択します。
   3. コンピュータのコマンドラインから、リポジトリをホストマシンのディレクトリに複製します。

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

   Important

   - このトピックでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。
   - *freertos* パスにスペース文字が含まれていると、構築が失敗する可能性があります。リポジトリをクローンまたはコピーするときは、作成するパスにスペース文字が含まれていないことを確認してください。
   - Microsoft Windows でのファイルパスの最大長は 260 文字です。FreeRTOS のダウンロードディレクトリパスが長くなると、構築が失敗する可能性があります。
   - ソースコードにはシンボリックリンクが含まれている可能性があるため、Windows を使用してアーカイブを抽出する場合は、次の操作を行う必要があります。
     - 開発者モードを有効にするか、または、
     - 管理者としてコンソールを使用します。

     この操作を行えば、Windows でアーカイブを抽出する際にシンボリックリンクを適切に作成できます。この操作を行わないと、シンボリックリンクは、そのパスがテキストとして含まれる、または空白の通常ファイルとして書き込まれます。詳細については、ブログの投稿「Symlinks in Windows 10!」を参照してください。

     Windows で Git を使用する場合は、開発者モードを有効にするか、以下を実行する必要があります。
     - 次のコマンドを使用して、`core.symlinks` を true に設定します。

```
git config -\-global core.symlinks true
```

- システムへの書き込みを行う git コマンド (git pull、git clone、git submodule update -\-init -\-recursive など) を使用する場合は、管理者としてコンソールを使用します。

4. *freertos* ディレクトリから、使用するブランチをチェックアウトします。

2. 開発環境をセットアップします。

   a. WinPCap の最新バージョンをインストールします。

   b. Microsoft Visual Studio をインストールします。

      Visual Studio バージョン 2017 および 2019 は動作することが確認されています。Visual Studio のすべてのエディションがサポートされます (Community、Professional、または Enterprise)。

      IDE に加えて、[Desktop development with C++] (C++ によるデスクトップ開発) コンポーネントをインストールします。次に、[オプション] で、最新の Windows 10 SDK をインストールします。

   c. アクティブな有線イーサネット接続が存在することを確認してください。

FreeRTOS デモプロジェクトを構築して実行する

> **Important**
>
> Microchip ATECC608A デバイスには、プロジェクトが初めて実行されるとき (`C_InitToken` 呼び出し時) にデバイスにロックされる 1 回だけの初期化があります。ただし、FreeRTOS デモプロジェクトおよびテストプロジェクトには別の設定があります。デモプロジェクト構成中にデバイスがロックされた場合、テストプロジェクトのすべてのテストを成功することはできません。

Visual Studio IDE で FreeRTOS デモプロジェクトを構築して実行するには

1. Visual Studio にプロジェクトをロードします。

[File] (ファイル) メニューで、[Open] (開く) を選択します。[File/Solution] (ファイル/ソリューション) を選択し、*freertos*\projects\microchip\ecc608a_plus_winsim\visual_studio\aws_demos\aws_demos.sln に移動してから [Open] (開く) を選択します。

2. デモプロジェクトをリターゲットします。

   指定されたデモプロジェクトは、Windows SDK によって異なりますが、Windows SDK バージョンは指定されていません。デフォルトでは、IDE でマシンに存在しない SDK バージョンを使用してデモのビルドが試行されることがあります。Windows SDK バージョンを設定するには、[aws_demos] を右クリックし、[Retarget Projects] (プロジェクトをリターゲット) を選択します。これにより、[Review Solution Actions] (ソリューションのアクションを確認) ウィンドウが開きます。マシンに存在する Windows SDK バージョン (ドロップダウンの初期値でかまいません) を選択して、[OK] を選択します。

3. プロジェクトを構築して実行します。

   [Build] (構築) メニューから [Build Solution] (ソリューションの構築) を選択し、ソリューションがエラーなしで構築されることを確認します。[Debug] (デバッグ)、[Start Debugging] (デバッグを開始) を選択し、プロジェクトを実行します。最初の実行では、デバイスインターフェイスを設定して再コンパイルする必要があります。詳細については、「ネットワークインターフェイスを設定する」を参照してください。

4. Microchip ATECC608A をプロビジョニングします。

   Microchip から、ATECC608A 部品のセットアップに役立ついくつかのスクリプトツールが提供されています。*freertos*\vendors\microchip\secure_elements\app\example_trust_chain_tool に移動し、README.md ファイルを開きます。

   README.md ファイルの手順に従って、デバイスをプロビジョニングします。このステップには、以下が含まれます。

   1. 認証機関を作成して登録します AWS。
   2. Microchip ATECC608A でキーを生成し、公開鍵とデバイスのシリアル番号をエクスポートする。
   3. デバイスの証明書を生成し、その証明書を に登録します AWS。
   4. CA 認定とデバイス証明書をデバイスにロードする。

5. FreeRTOS サンプルを構築して実行します。

デモプロジェクトを再実行します。これで、接続に成功します。

トラブルシューティング

一般的なトラブルシューティング情報については、「トラブルシューティングの開始方法」を参照してください。

Espressif ESP32-DevKitC と ESP-WROVER-KIT の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

> **Note**
>
> 独自の Espressif IDF プロジェクト内で FreeRTOS モジュラーライブラリとデモを統合する方法の詳細については、ESP32-C3 プラットフォーム向けの注目のリファレンス統合を参照してください。

このチュートリアルでは、ESP32-WROOM-32、ESP32-SOLO-1、または ESP-WROVER モジュールおよび ESP-WROVER-KIT-VB を搭載した Espressif ESP32-DevKitC の使用を開始するための手順について説明します。Partner AWS Device カタログでパートナーから購入するには、次のリンクを使用します。

- ESP32-WROOM-32 DevKitC
- ESP32-SOLO-1
- ESP32-WROVER-KIT

開発ボードのこれらのバージョンが FreeRTOS でサポートされています。

これらのボードの最新バージョンの詳細については、Espressif ウェブサイトの「ESP32-DevKitC V4」または「ESP-WROVER-KIT v4.1」を参照してください。

> **Note**
>
> 現在、ESP32-WROVER-KIT および ESP DevKitC の FreeRTOS ポートは、対称型マルチプロセッシング (SMP) 機能をサポートしていません。

概要

このチュートリアルでは次のステップを説明します。

1. ボードをホストマシンに接続します。
2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
5. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

前提条件

Espressif ボードで FreeRTOS の使用を開始する前に、 AWS アカウントとアクセス許可を設定する必要があります。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルー

トユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

  アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。
- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。
- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

はじめに

> **Note**
>
> このチュートリアルの Linux コマンドでは、Bash シェルを使用する必要があります。

1. Espressif ハードウェアを設定します。

   - ESP32-DevKitC 開発ボードハードウェアの設定の詳細については、ESP32-DevKitC V4 入門ガイドを参照してください。
   - ESP-WROVER-KIT 開発ボードハードウェアの設定の詳細については、ESP-WROVER-KIT V4.1 入門ガイドを参照してください。

   > **Important**
   >
   > Espressif ガイドの開始方法セクションに到達したらそこで止まり、このページの手順に戻ります。

2. GitHub から Amazon FreeRTOS をダウンロードします。(手順については、README.md ファイルを参照してください。)
3. 開発環境をセットアップします。

   ボードと通信するには、ツールチェーンをインストールする必要があります。Espressif には、ボード用のソフトウェアを開発するための ESP-IDF が用意されています。ESP-IDF には独自のバージョンの FreeRTOS カーネルがコンポーネントとして統合されているため、Amazon FreeRTOS には、FreeRTOS カーネルが除去された、カスタムバージョンの ESP-IDF v4.2 が含まれています。これにより、コンパイル時にファイルが重複する問題が修正されます。Amazon FreeRTOS に含まれているカスタムバージョンの ESP-IDF v4.2 を使用するには、ホストマシンのオペレーティングシステムに応じて以下の手順を実行します。

   Windows

   1. ESP-IDF の Windows 用の汎用オンラインインストーラをダウンロードします。
   2. 汎用オンラインインストーラを実行します。

3. ESP-IDF をダウンロードまたは使用する手順に進んだら、[既存の ESP-IDF ディレクトリを使用] を選択し、[既存の ESP-IDF ディレクトリを選択] を *freertos*/vendors/espressif/esp-idf に設定します。
4. インストールを完了します。

macOS

1. 「Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for macOS」の手順を実行します。

> **Important**
> 次のステップの下にある「ESP-IDF の取得」の手順に到達したらそこで止まり、このページの手順に戻ります。

2. コマンドラインウィンドウを開きます。
3. FreeRTOS ダウンロードディレクトリに移動し、次のスクリプトを実行して、お使いのプラットフォーム用の espressif ツールチェーンをダウンロードしてインストールします。

```
vendors/espressif/esp-idf/install.sh
```

4. 次のコマンドを使用して、ESP-IDF ツールチェーンツールをターミナルのパスに追加します。

```
source vendors/espressif/esp-idf/export.sh
```

Linux

1. 「Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for Linux」の手順を実行します。

> **Important**
> 次のステップの下にある「ESP-IDF の取得」の手順に到達したらそこで止まり、このページの手順に戻ります。

2. コマンドラインウィンドウを開きます。

3. FreeRTOS ダウンロードディレクトリに移動し、次のスクリプトを実行して、お使いのプラットフォーム用の espressif ツールチェーンをダウンロードしてインストールします。

```
vendors/espressif/esp-idf/install.sh
```

4. 次のコマンドを使用して、ESP-IDF ツールチェーンツールをターミナルのパスに追加します。

```
source vendors/espressif/esp-idf/export.sh
```

4. シリアル接続を確立します。

   a. ホストマシンと ESP32-DevKitC の間にシリアル接続を確立するには、CP210x USB を UART Bridge VCP ドライバーにインストールする必要があります。これらのドライバーは Silicon Labs からダウンロードできます。

      ホストマシンと ESP32-WROVER-KIT の間にシリアル接続を確立するには、FTDI 仮想 COM ポートドライバーをインストールする必要があります。このドライバーは FTDI からダウンロードできます。

   b. ESP32 でシリアル接続を確立するステップを実行します。

   c. シリアル接続を確立したら、ボードとの接続用のシリアルポートをメモしておきます。デモをフラッシュするにはこれが必要です。

FreeRTOS デモアプリケーションを設定する

このチュートリアルでは、FreeRTOS 設定ファイルは *freertos*/vendors/espressif/boards/*board-name*/aws_demos/config_files/FreeRTOSConfig.h にあります。(例えば、AFR_BOARD espressif.esp32_devkitc を選択した場合、設定ファイルは *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/FreeRTOSConfig.h にあります。)

1. macOS または Linux を実行している場合、ターミナルプロンプトを開きます。Windows を実行している場合は、「ESP-IDF 4.x CMD」アプリ (ESP-IDF ツールチェーンのインストール時にこのオプションを含めた場合) または「コマンドプロンプト」アプリ (オプションを含めなかった場合) を開きます。
2. Python3 がインストールされていることを確認するには、以下を実行します。

```
python --version
```

インストールされているバージョンが表示されます。Python 3.0.1 以降がインストールされていない場合は、Python ウェブサイトからインストールできます。

3. AWS IoT コマンドを実行するには AWS 、 コマンドラインインターフェイス (CLI) が必要です。Windows を実行している場合は、 `easy_install awscli` コマンドを使用して AWS 、「Command」または「ESP-IDF 4.x CMD」アプリに CLI をインストールします。

   macOS または Linux を実行している場合は、「 AWS CLI のインストール」を参照してください。

4. 実行

   ```
   aws configure
   ```

   アクセスキー ID AWS 、シークレット AWS アクセスキー、デフォルト AWS リージョンを使用して CLI を設定します。詳細については、AWS CLI の設定を参照してください。

5. 次のコマンドを使用して AWS SDK for Python (boto3) をインストールします。

   - Windows の場合、「コマンド」または「ESP-IDF 4.x CMD」アプリで、以下を実行します。

     ```
     pip install boto3 --user
     ```

     > **Note**
     >
     > 詳細については、Boto3 ドキュメントを参照してください。

   - macOS または Linux の場合、以下を実行します。

     ```
     pip install tornado nose --user
     ```

     続いて以下を実行します。

     ```
     pip install boto3 --user
     ```

FreeRTOS には、 AWS IoTに接続するための Espressif ボードのセットアップを容易にする SetupAWS.py スクリプトが含まれています。このスクリプトを設定するには、*freertos*/tools/aws_config_quick_start/configure.json を開いて以下の属性を設定します。

afr_source_dir

コンピュータの *freertos* ディレクトリへの完全なパス。このパスの指定にスラッシュを使用していることを確認します。

thing_name

ボードを表す AWS IoT モノに割り当てる名前。

wifi_ssid

Wi-Fi ネットワークの SSID。

wifi_password

Wi-Fi ネットワークのパスワード。

wifi_security

Wi-Fi ネットワークのセキュリティタイプ。

次のセキュリティタイプが有効です。

- eWiFiSecurityOpen (オープン、セキュリティなし)
- eWiFiSecurityWEP (WEP セキュリティ)
- eWiFiSecurityWPA (WPA セキュリティ)
- eWiFiSecurityWPA2 (WPA2 セキュリティ)

6. 設定スクリプトを実行します。

   a. macOS または Linux を実行している場合、ターミナルプロンプトを開きます。Windows を実行している場合は、「ESP-IDF 4.x CMD」アプリまたは「コマンド」アプリを開きます。

   b. *freertos*/tools/aws_config_quick_start ディレクトリに移動して、以下を実行します。

```
python SetupAWS.py setup
```

スクリプトは、次を実行します。

- IoT のモノ、証明書およびポリシーを作成します。
- 証明書に IoT ポリシーを、 AWS IoT のモノに証明書をアタッチします。
- `aws_clientcredential.h` エンドポイント、Wi-Fi SSID、および認証情報を AWS IoT ファイルに追加します。
- 証明書とプライベートキーをフォーマットして `aws_clientcredential_keys.h` ヘッダーファイルに書き込みます。

> **Note**
>
> 証明書はデモ目的でのみハードコードされています。本番稼動レベルのアプリケーションでは、これらのファイルを安全な場所に保存する必要があります。

`SetupAWS.py` の詳細については、*`freertos`*`/tools/aws_config_quick_start` ディレクトリにある「`README.md`」を参照してください。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソールに移動します。
2. ナビゲーションペインで、[テスト] を選択し、次に [MQTT テストクライアント] を選択します。
3. [Subscription topic] (トピックのサブスクリプション) で *`your-thing-name`*`/example/topic` と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

idf.py スクリプトを使用して FreeRTOS デモプロジェクトを構築、フラッシュ、実行する

Espressif の IDF ユーティリティ (idf.py) 使用してプロジェクトを構築し、バイナリをデバイスにフラッシュすることができます。

> **Note**
>
> 一部のセットアップでは、次の例のように、idf.py でポートオプション "-p port-name" を使用して正しいポートを指定する必要があります。
>
> ```
> idf.py -p /dev/cu.usbserial-00101301B flash
> ```

Windows、Linux、macOS で FreeRTOS を構築してフラッシュする (ESP-IDF v4.2)

1. FreeRTOS ダウンロードディレクトリのルートに移動します。
2. コマンドラインウィンドウで、次のコマンドを入力し、ESP-IDF ツールをターミナルのパスに追加します。

   Windows (「コマンド」アプリ)

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Windows (「ESP-IDF 4.x CMD」アプリ)

   (これはアプリを開いた時点で既に完了しています。)

   Linux/macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```
3. 次のコマンドを使用して build ディレクトリで cmake を設定し、ファームウェアイメージを構築します。

   ```
   idf.py -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 build
   ```

   次のような出力が表示されます。

   ```
   Running cmake in directory /path/to/hello_world/build
   Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
   ```

```
   Warn about uninitialized values.
   -- Found Git: /usr/bin/git (found version "2.17.0")
   -- Building empty aws_iot component due to configuration
   -- Component names: ...
   -- Component paths: ...

   ... (more lines of build system output)

   [527/527] Generating hello-world.bin
   esptool.py v2.3.1

   Project build complete. To flash, run this command:
   ../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600
 write_flash --flash_mode dio --flash_size detect --flash_freq 40m 0x10000 build/
hello-world.bin  build 0x1000 build/bootloader/bootloader.bin 0x8000 build/
partition_table/partition-table.bin
   or run 'idf.py -p PORT flash'
```

エラーが起きなければ、構築によってファームウェアバイナリの .bin ファイルが生成されます。

4. 次のコマンドを使用して、開発ボードのフラッシュメモリを消去します。

```
idf.py erase_flash
```

5. idf.py スクリプトを使用して、アプリケーションバイナリをボードにフラッシュします。

```
idf.py flash
```

6. 次のコマンドを使用して、ボードのシリアルポートからの出力をモニタリングします。

```
idf.py monitor
```

> **Note**
>
> 以下の例のように、これらのコマンドを組み合わせることができます。
>
> ```
> idf.py erase_flash flash monitor
> ```

特定のホストマシンのセットアップでは、次の例のように、ボードをフラッシュするときにポートを指定する必要があります。

```
idf.py erase_flash flash monitor -p /dev/ttyUSB1
```

CMake で FreeRTOS を構築してフラッシュする

コードを構築して実行するために IDF SDK によって提供される `idf.py` スクリプトに加えて、CMake を使用してプロジェクトを構築することもできます。現在、Unix Makefiles または Ninja ビルドシステムのいずれかがサポートされています。

プロジェクトを構築してフラッシュするには

1. コマンドラインウィンドウで、FreeRTOS ダウンロードディレクトリのルートに移動します。
2. 次のスクリプトを実行して、ESP-IDF ツールをシェルのパスに追加します。

   Windows

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Linux/macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```
3. 次のコマンドを入力して、ビルドファイルを生成します。

   Unix Makefiles を使用

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 -S . -
   B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0
   ```

   Ninja を使用

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32_wrover_kit -DCOMPILER=xtensa-esp32 -S . -
   B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0 -GNinja
   ```
4. プロジェクトをビルドします。

Unix Makefiles を使用

```
make -C ./YOUR_BUILD_DIRECTORY -j8
```

Ninja を使用

```
ninja -C ./YOUR_BUILD_DIRECTORY -j8
```

5. フラッシュを消去してから、ボードをフラッシュします。

Unix Makefiles を使用

```
make -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
make -C ./YOUR_BUILD_DIRECTORY flash
```

Ninja を使用

```
ninja -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
ninja -C ./YOUR_BUILD_DIRECTORY flash
```

Bluetooth Low Energy デモを実行する

FreeRTOS は Bluetooth Low Energy ライブラリ 接続をサポートしています。

Bluetooth Low Energy で FreeRTOS デモプロジェクトを実行するには、iOS または Android のモバイルデバイスで FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションを実行する必要があります。

FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションをセットアップするには

1. モバイルプラットフォーム用の SDK をホストコンピュータにダウンロードしてインストールするには、「FreeRTOS Bluetooth デバイス用の Mobile SDK」の手順に従います。
2. モバイルデバイスにデモモバイルアプリケーションをセットアップするには、「FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション」の手順に慕います。

ボードで MQTT over Bluetooth Low Energy デモを実行する方法については、MQTT over Bluetooth Low Energyを参照してください。

ボードで Wi-Fi プロビジョニングデモを実行する方法については、Wi-Fi プロビジョニングを参照してください。

ESP32 用の独自の CMake プロジェクトで FreeRTOS を使用する

独自の CMake プロジェクトで FreeRTOS を使用したい場合は、それをサブディレクトリとして設定し、アプリケーションと一緒に構築することができます。まず、GitHub から FreeRTOS のコピーを取得します。それを、次のコマンドを使用して Git サブモジュールとして設定することもできます。これにより、将来の更新が容易になります。

```
git submodule add -b release https://github.com/aws/amazon-freertos.git freertos
```

新しいバージョンがリリースされた場合、これらのコマンドを使用してローカルコピーを更新できます。

```
# Pull the latest changes from the remote tracking branch.
git submodule update --remote -- freertos
```

```
# Commit the submodule change because it is pointing to a different revision now.
git add freertos
```

```
git commit -m "Update FreeRTOS to a new release"
```

プロジェクトのディレクトリ構造が次のものであると仮定します。

```
- freertos (the copy that you obtained from GitHub or the AWS IoT console)
- src
  - main.c (your application code)
- CMakeLists.txt
```

以下に、FreeRTOS を使用してアプリケーションを構築するために使用できる最上位 `CMakeLists.txt` ファイルの例を示します。

```
cmake_minimum_required(VERSION 3.13)

project(freertos_examples)
```

```
# Tell IDF build to link against this target.
set(IDF_EXECUTABLE_SRCS "<complete_path>/src/main.c")
set(IDF_PROJECT_EXECUTABLE my_app)

# Add FreeRTOS as a subdirectory. AFR_BOARD tells which board to target.
set(AFR_BOARD espressif.esp32_devkitc CACHE INTERNAL "")
add_subdirectory(freertos)

# Link against the mqtt library so that we can use it. Dependencies are transitively
# linked.
target_link_libraries(my_app PRIVATE AFR::core_mqtt)
```

プロジェクトをビルドするには、次の CMake コマンドを実行します。ESP32 コンパイラが PATH 環境変数にあることを確認してください。

```
cmake -S . -B build-directory -DCMAKE_TOOLCHAIN_FILE=freertos/tools/cmake/toolchains/
xtensa-esp32.cmake -GNinja
```

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

アプリケーションをボードにフラッシュするには、次のコマンドを実行します。

```
cmake --build build-directory --target flash
```

FreeRTOS のコンポーネントを使用する

CMake を実行した後、サマリー出力で使用可能なすべてのコンポーネントを見つけることができます。次の例のようになります。

```
====================Configuration for FreeRTOS====================
  Version:                 202107.00
  Git version:             202107.00-g79ad6defb

Target microcontroller:
  vendor:                  Espressif
  board:                   ESP32-DevKitC
  description:             Development board produced by Espressif that comes in two
                           variants either with ESP-WROOM-32 or ESP32-WROVER module
  family:                  ESP32
  data ram size:           520KB
  program memory size:     4MB
```

```
Host platform:
  OS:                       Linux-4.15.0-66-generic
  Toolchain:                xtensa-esp32
  Toolchain path:           /opt/xtensa-esp32-elf
  CMake generator:          Ninja

FreeRTOS modules:
  Modules to build:         backoff_algorithm, common, common_io, core_http,
                            core_http_demo_dependencies, core_json, core_mqtt,
                            core_mqtt_agent, core_mqtt_agent_demo_dependencies,
                            core_mqtt_demo_dependencies, crypto, defender, dev_mode_key_
                            provisioning, device_defender, device_defender_demo_
                            dependencies, device_shadow,
 device_shadow_demo_dependencies,
                            freertos_cli_plus_uart, freertos_plus_cli, greengrass,
                            http_demo_helpers, https, jobs, jobs_demo_dependencies,
                            kernel, logging, mqtt, mqtt_agent_interface, mqtt_demo_
                            helpers, mqtt_subscription_manager, ota, ota_demo_
                            dependencies, ota_demo_version, pkcs11, pkcs11_helpers,
                            pkcs11_implementation, pkcs11_utils, platform,
 secure_sockets,
                            serializer, shadow, tls, transport_interface_secure_sockets,
                            wifi
   Enabled by user:         common_io, core_http_demo_dependencies, core_json,
                            core_mqtt_agent_demo_dependencies, core_mqtt_demo_
                            dependencies, defender, device_defender,
 device_defender_demo_
                            dependencies, device_shadow,
 device_shadow_demo_dependencies,
                            freertos_cli_plus_uart, freertos_plus_cli, greengrass,
 https,
                            jobs, jobs_demo_dependencies, logging,
 ota_demo_dependencies,
                            pkcs11, pkcs11_helpers, pkcs11_implementation, pkcs11_utils,
                            platform, secure_sockets, shadow, wifi
   Enabled by dependency:   backoff_algorithm, common, core_http, core_mqtt,
                            core_mqtt_agent, crypto, demo_base,
 dev_mode_key_provisioning,
                            freertos, http_demo_helpers, kernel, mqtt, mqtt_agent_
                            interface, mqtt_demo_helpers, mqtt_subscription_manager,
 ota,
                            ota_demo_version, pkcs11_mbedtls, serializer, tls,
                            transport_interface_secure_sockets, utils
   3rdparty dependencies:   jsmn, mbedtls, pkcs11, tinycbor
```

```
  Available demos:         demo_cli_uart, demo_core_http, demo_core_mqtt,
 demo_core_mqtt_
                          agent, demo_device_defender, demo_device_shadow,
                          demo_greengrass_connectivity, demo_jobs, demo_ota_core_http,
                          demo_ota_core_mqtt, demo_tcp
  Available tests:
===========================================================================
```

Modules to build リストから任意のコンポーネントを参照できます。それらをアプリケーションにリンクするには、名前の前に AFR:: 名前空間を置きます。例えば AFR::core_mqtt、AFR::ota などです。

ESP-IDF を使用したカスタムコンポーネントの追加

ESP-IDF の使用時に、さらにコンポーネントを追加できます。たとえば、example_component というコンポーネントを追加し、プロジェクトは次のようになります。

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

コンポーネントの CMakeLists.txt ファイルの例を次に示します。

```
add_library(example_component src/example_component.c)
target_include_directories(example_component PUBLIC include)
```

次に、最上位の CMakeLists.txt ファイルで、add_subdirectory(freertos) の直後に次の行を挿入してコンポーネントを追加します。

```
add_subdirectory(component/example_component)
```

次に、コンポーネントを含めるように target_link_libraries を変更します。

```
target_link_libraries(my_app PRIVATE AFR::core_mqtt PRIVATE example_component)
```

このコンポーネントは、デフォルトでアプリケーションコードに自動的にリンクされるようになりました。ヘッダーファイルを含めて、定義する関数を呼び出せるようになりました。

FreeRTOS の設定を上書きする

現在、FreeRTOS ソースツリーの外で構成を再定義するための明確なアプローチはありません。デフォルトでは、CMake による検索対象は *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/ ディレクトリと *freertos*/demos/include/ ディレクトリです。ただし、回避策を使用して、最初に他のディレクトリを検索するようにコンパイラに指示できます。例えば、FreeRTOS 設定用に別のフォルダを追加できます。

```
- freertos
- freertos-configs
  - aws_clientcredential.h
  - aws_clientcredential_keys.h
  - iot_mqtt_agent_config.h
  - iot_config.h
- components
- src
- CMakeLists.txt
```

freertos-configs の下にあるファイルは、*freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/ ディレクトリと *freertos*/demos/include/ ディレクトリからコピーされます。次に、最上位レベルの CMakeLists.txt ファイルで、次の行を add_subdirectory(freertos) の前に追加し、このディレクトリをコンパイラーが最初に検索するようにします。

```
include_directories(BEFORE freertos-configs)
```

ESP-IDF 用の独自の sdkconfig を提供する

独自の sdkconfig.default を提供したい場合は、コマンドラインから CMake 変数 IDF_SDKCONFIG_DEFAULTS を設定することができます。

```
cmake -S . -B build-directory -DIDF_SDKCONFIG_DEFAULTS=path_to_your_sdkconfig_defaults
 -DCMAKE_TOOLCHAIN_FILE=freertos/tools/cmake/toolchains/xtensa-esp32.cmake -GNinja
```

独自の sdkconfig.default ファイルの場所を指定しない場合、FreeRTOS では *freertos*/vendors/espressif/boards/esp32/aws_demos/sdkconfig.defaults にあるデフォルトのファイルが使用されます。

詳細については、「Espressif API Reference」の「Project Configuration」を参照してください。コンパイルが正常に完了した後に問題が発生する場合は、そのページの「Deprecated options and their replacements」のセクションを参照してください。

概要

example_component という名前のコンポーネントを持つプロジェクトがあり、いくつかの設定を上書きする場合、最上位レベルの CMakeLists.txt ファイルの完全な例を次に示します。

```
cmake_minimum_required(VERSION 3.13)

project(freertos_examples)

set(IDF_PROJECT_EXECUTABLE my_app)
set(IDF_EXECUTABLE_SRCS "src/main.c")

# Tell IDF build to link against this target.
set(IDF_PROJECT_EXECUTABLE my_app)

# Add some extra components. IDF_EXTRA_COMPONENT_DIRS is a variable used by ESP-IDF
# to collect extra components.
get_filename_component(
    EXTRA_COMPONENT_DIRS
    "components/example_component" ABSOLUTE
)
list(APPEND IDF_EXTRA_COMPONENT_DIRS ${EXTRA_COMPONENT_DIRS})

# Override the configurations for FreeRTOS.
include_directories(BEFORE freertos-configs)

# Add FreeRTOS as a subdirectory. AFR_BOARD tells which board to target.
set(AFR_BOARD espressif.esp32_devkitc CACHE INTERNAL "")
add_subdirectory(freertos)

# Link against the mqtt library so that we can use it. Dependencies are transitively
# linked.
target_link_libraries(my_app PRIVATE AFR::core_mqtt)
```

トラブルシューティング

- macOS を使用していてオペレーティングシステムが ESP-WROVER-KIT を認識しない場合は、D2XX ドライバーがインストールされていないことを確認してください。ドライバーをアンインストールするには、FTDI Drivers Installation Guide for macOS X の手順に従います。
- ESP-IDF によって提供された (および make monitor を使用して呼び出された) モニタユーティリティを使用して、アドレスをデコードできます。そのため、アプリケーションが停止した場合に意味のあるバックトレースを取得するのに役立ちます。詳細については、Espressif ウェブサイトの自動アドレスデコーディングを参照してください。
- GDBstub を gdb との通信用に有効にすることができます。特別な JTAG ハードウェアは必要ありません。詳細については、Espressif ウェブサイトの GDBStub を使用して GDB を起動するを参照してください。
- JTAG ハードウェアベースのデバッグが必要な場合に OpenOCD ベースの環境をセットアップする方法の詳細については、Espressif ウェブサイトの JTAG のデバッグを参照してください。
- macOS で `pip` を使用して `pyserial` をインストールできない場合、pyserial ウェブサイト からダウンロードします。
- ボードが継続的にリセットされる場合は、ターミナルで次のコマンドを入力して、フラッシュの消去を試してください。

```
make erase_flash
```

- `idf_monitor.py` を実行するときにエラーが表示された場合は、Python 2.7 を使用します。
- ESP-IDF からの必須ライブラリは FreeRTOS に含まれています。外部でダウンロードする必要はありません。`IDF_PATH` 環境変数が設定されている場合、FreeRTOS を構築する前にクリアすることをお勧めします。
- Windows では、プロジェクトのビルドに 3 ~ 4 分かかる場合があります。構築時間を減らすために、make コマンドで `-j4` スイッチを使用できます。

```
make flash monitor -j4
```

- デバイスに への接続に問題がある場合は AWS IoT、 `aws_clientcredential.h` ファイルを開き、設定変数が ファイルで正しく定義されていることを確認します。`clientcredentialMQTT_BROKER_ENDPOINT[]` は のようになります`1234567890123-ats.iot.us-east-1.amazonaws.com`。
- 「ESP32 用の独自の CMake プロジェクトで FreeRTOS を使用する」の手順を実行していて、リンカーから未定義の参照エラーが返された場合は、通常、依存ライブラリまたはデモがないこと

を示します。これらを追加するには、標準の CMake 関数 `target_link_libraries` を使用して `CMakeLists.txt` ファイル (ルートディレクトリの下) を更新します。

- ESP-IDF v4.2 は、xtensa\-esp32\-elf\-gcc 8\.2\.0\. ツールチェーンの使用をサポートしています。以前のバージョンの Xtensa ツールチェーンを使用している場合、必要なバージョンをダウンロードしてください。
- ESP-IDF v4.2 で Python の依存関係が満たされていないことについて、次のようなエラーログが表示された場合:

```
The following Python requirements are not satisfied:
  click>=5.0
  pyserial>=3.0
  future>=0.15.2
  pyparsing>=2.0.3,<2.4.0
  pyelftools>=0.22
  gdbgui==0.13.2.0
  pygdbmi<=0.9.0.2
  reedsolo>=1.5.3,<=1.5.4
  bitstring>=3.1.6
  ecdsa>=0.16.0
  Please follow the instructions found in the "Set up the tools" section of ESP-IDF
 Getting Started Guide
```

次の Python コマンドを使用して、プラットフォームに Python の依存関係をインストールします。

```
root/vendors/espressif/esp-idf/requirements.txt
```

トラブルシューティングの詳細については、「トラブルシューティングの開始方法」を参照してください。

デバッグ

Espressif ESP32-DevKitC および ESP-WROVER-KIT のデバッグコード (ESP-IDF v4.2)

このセクションでは、ESP-IDF v4.2 を使用して Espressif ハードウェアをデバッグする方法について説明します。JTAG to USB ケーブルが必要です。USB to MPSSE ケーブルを使用します (例: FTDI C232HM-DDHSL-0)。

ESP-DevKitC JTAG セットアップ

FTDI C232HM-DDHSL-0 ケーブルの場合、これらは ESP32 DevkitC への接続です。

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

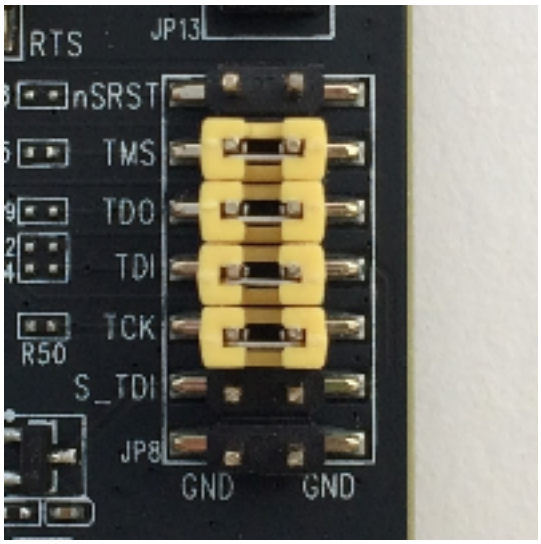
ESP-WROVER-KIT JTAG セットアップ

FTDI C232HM-DDHSL-0 ケーブルの場合、これらは ESP32-WROVER-KIT への接続です。

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

これらのテーブルは、FTDI C232HM-DDHSL-0 データシートから開発されました。詳細については、データシートの「C232HM MPSSE Cable Connection and Mechanical Details」セクションを参照してください。

ESP-WROVER-KIT で JTAG を有効にするには、次に示すように TMS、TDO、TDI、TCK、S_TDI のピンにジャンパーを配置します。



Windows でのデバッグ (ESP-IDF v4.2)

Windows でのデバッグをセットアップするには

1. FTDI C232HM-DDHSL-0 の USB 側をコンピュータに接続し、他方の側を Espressif ESP32-DevKitC および ESP-WROVER-KIT のデバッグコード (ESP-IDF v4.2) の説明に従って接続します。FTDI C232HM-DDHSL-0 デバイスは、[Universal Serial Bus Controllers] (ユニバーサルシリアルバスコントローラー) の [Device Manager] (デバイスマネージャー) に表示されます。
2. ユニバーサルシリアルバスデバイスのリストで、[C232HM-DDHSL-0] デバイスを右クリックし、[Properties] (プロパティ) を選択します。

   > **Note**
   >
   > このデバイスは、[USB Serial Port] (USB シリアルポート) として表示される場合があります。

   プロパティウィンドウで [Details] (詳細) タブを選択して、デバイスのプロパティを表示します。デバイスが表示されない場合は、Windows driver for FTDI C232HM-DDHSL-0 をインストールします。
3. [Details] (詳細) タブで、[Property] (プロパティ) を選択し、[Hardware IDs] (ハードウェア ID) を選択します。[Value] (値) フィールドに、次のような内容が表示されます。

   ```
   FTDIBUS\COMPORT&VID_0403&PID_6014
   ```

   この例では、ベンダー ID は 0403 で、製品 ID は 6014 です。

   これらの ID が `projects/espressif/esp32/make/aws_demos/esp32_devkitj_v1.cfg` の ID に一致していることを確認します。ID は `ftdi_vid_pid` で始まる行で指定されており、その後にベンダー ID と製品 ID が続きます。

   ```
   ftdi_vid_pid 0x0403 0x6014
   ```
4. OpenOCD for Windows をダウンロードします。
5. ファイルを `C:\` に解凍して、システムパスに `C:\openocd-esp32\bin` を追加します。

6. OpenOCD には libusb が必要です。これは、Windows にデフォルトではインストールされません。libusb をインストールするには次のステップを実行します。

   a. zadig.exe をダウンロードします。

   b. `zadig.exe` を実行します。[Options] (オプション) メニューから、[List All Devices] (すべてのデバイスをリストする) を選択します。

   c. ドロップダウンメニューから [C232HM-DDHSL-0] を選択します。

   d. ターゲットドライバーフィールドの緑の矢印の右側で [WinUSB] を選択します。

   e. ターゲットドライバーフィールドの下のリストで、矢印を選択して [Install Driver] (ドライバーのインストール) を選択します。[Replace Driver] (ドライバーの置換) を選択します。

7. コマンドプロンプトを開き、FreeRTOS ダウンロードディレクトリのルートに移動し、次のコマンドを実行します。

   ```
   idf.py openocd
   ```

   このコマンドプロンプトを開いたままにしておきます。

8. 新しいコマンドプロンプトを開き、FreeRTOS ダウンロードディレクトリのルートに移動して以下を実行します。

   ```
   idf.py flash monitor
   ```

9. 別のコマンドプロンプトを開き、FreeRTOS ダウンロードディレクトリのルートに移動し、ボードでデモの実行が開始されるまで待ちます。このときに、以下を実行します。

   ```
   idf.py gdb
   ```

   このプログラムは `main` 関数で停止する必要があります。

   > **Note**
   >
   > ESP32 では、最大 2 個のブレークポイントがサポートされています。

macOS (ESP-IDF v4.2) でのデバッグ

1. FTDI driver for macOS をダウンロードします。

2. OpenOCD をダウンロードします。
3. ダウンロードした .tar ファイルを抽出して、.bash_profile のパスを OCD_INSTALL_DIR/openocd-esp32/bin に設定します。
4. 次のコマンドを使用して macOS に libusb をインストールします。

```
brew install libusb
```

5. 次のコマンドを使用してシリアルポートドライバーをアンロードします。

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

6. 次のコマンドを使用してシリアルポートドライバーをアンロードします。

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

7. macOS 10.9 以降のバージョンを実行している場合は、次のコマンドを使用して Apple の FTDI ドライバーをアンロードします。

```
sudo kextunload -b com.apple.driver.AppleUSBFTDI
```

8. 次のコマンドを使用して FTDI ケーブルの製品 ID とベンダー ID を取得します。アタッチされた USB デバイスが表示されます。

```
system_profiler SPUSBDataType
```

system_profiler からの出力は次のようになります。

```
    DEVICE:

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

9. projects/espressif/esp32/make/aws_demos/esp32_devkitj_v1.cfg ファイルを開きます。デバイスのベンダー ID および製品 ID は ftdi_vid_pid で始まる行で指定されています。前のステップの system_profiler 出力の ID に一致するように ID を変更します。
10. ターミナルウィンドウを開き、FreeRTOS ダウンロードディレクトリのルートに移動して、以下のコマンドを使用して OpenOCD を実行します。

```
idf.py openocd
```

ターミナルウィンドウは開いたままにします。

11. 新しいターミナルを開き、次のコマンドを使用して FTDI シリアルポートドライバーをロードします。

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

12. FreeRTOS ダウンロードディレクトリのルートに移動し、以下を実行します。

```
idf.py flash monitor
```

13. 別の新しいターミナルを開き、FreeRTOS のダウンロードディレクトリのルートに移動し、以下を実行します。

```
idf.py gdb
```

`main` でこのプログラムを停止する必要があります。

Linux でのデバッグ (ESP-IDF v4.2)

1. OpenOCD をダウンロードします。tarball を抽出し、readme ファイルのインストール手順に従ってください。
2. 次のコマンドを使用して Linux に libusb をインストールします。

```
sudo apt-get install libusb-1.0
```

3. ターミナルを開いて ls -l /dev/ttyUSB* と入力し、コンピュータに接続されているすべての USB デバイスを一覧表示します。これにより、ボードの USB ポートがオペレーティングシステムによって認識されているかどうかを確認できます。次のような出力が表示されます。

```
   $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. サインオフしてからサインインし、ボードの電源を入れ直して変更を有効にします。ターミナルプロンプトで、USB デバイスを一覧表示します。グループ所有者が dialout から plugdev に変化していることを確認します。

```
   $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
```

   数の少ない番号を持つ /dev/ttyUSBn インターフェイスは、JTAG 通信に使用されます。もう 1 つのインターフェイスは ESP32 のシリアルポート (UART) にルーティングされ、コードを ESP32 のフラッシュメモリにアップロードするために使用されます。

5. ターミナルウィンドウで FreeRTOS ダウンロードディレクトリのルートに移動して、以下のコマンドを使用して OpenOCD を実行します。

```
idf.py openocd
```

6. 別のターミナルを開き、FreeRTOS ダウンロードディレクトリのルートに移動し、以下のコマンドを実行します。

```
idf.py flash monitor
```

7. 別のターミナルを開き、FreeRTOS ダウンロードディレクトリのルートに移動し、以下のコマンドを実行します。

```
idf.py gdb
```

   main() でこのプログラムを停止する必要があります。

Espressif ESP32-WROOM-32SE の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の

Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

Note

- 独自の Espressif IDF プロジェクト内で FreeRTOS モジュラーライブラリとデモを統合する方法の詳細については、ESP32-C3 プラットフォーム向けの注目のリファレンス統合を参照してください。
- 現在、ESP32-WROOM-32SE の FreeRTOS ポートは、対称型マルチプロセッシング (SMP) 機能をサポートしていません。

このチュートリアルでは、Espressif ESP32-WROOM-32SE の使用を開始する方法について説明します。パートナーデバイスカタログで AWS パートナーから購入するには、「ESP32-WROOM-32SE」を参照してください。

概要

このチュートリアルでは次のステップを説明します。

1. ボードをホストマシンに接続します。
2. ホストマシンにソフトウェアをインストールし、マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
5. シリアル接続を使用することによって、実行中のアプリケーションをモニタリングおよびデバッグします。

前提条件

Espressif ボードで FreeRTOS の使用を開始する前に、 AWS アカウントとアクセス許可を設定する必要があります。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

   にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。

- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。

- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

はじめに

> **Note**
>
> このチュートリアルの Linux コマンドでは、Bash シェルを使用する必要があります。

1. Espressif ハードウェアを設定します。

   ESP32-WROOM-32SE 開発ボードハードウェアの設定の詳細については、ESP32-DevKitC V4 入門ガイドを参照してください。

   > **Important**
   >
   > ガイドの「Installation Step by Step」セクションまで進んだら、ステップ 4 (環境変数の設定) まで実行します。ステップ 4 を完了したら停止し、残りのステップをここで実行します。

2. GitHub から Amazon FreeRTOS をダウンロードします。(手順については、README.md ファイルを参照してください。)
3. 開発環境をセットアップします。

   ボードと通信するには、ツールチェーンをインストールする必要があります。Espressif には、ボード用のソフトウェアを開発するための ESP-IDF が用意されています。ESP-IDF には独自

のバージョンの FreeRTOS カーネルがコンポーネントとして統合されているため、Amazon FreeRTOS には、FreeRTOS カーネルが除去された、カスタムバージョンの ESP-IDF v4.2 が含まれています。これにより、コンパイル時にファイルが重複する問題が修正されます。Amazon FreeRTOS に含まれているカスタムバージョンの ESP-IDF v4.2 を使用するには、ホストマシンのオペレーティングシステムに応じて以下の手順を実行します。

Windows

1. ESP-IDF の Windows 用の汎用オンラインインストーラをダウンロードします。
2. 汎用オンラインインストーラを実行します。
3. ESP-IDF をダウンロードまたは使用する手順に進んだら、[既存の ESP-IDF ディレクトリを使用] を選択し、[既存の ESP-IDF ディレクトリを選択] を *freertos*/vendors/espressif/esp-idf に設定します。
4. インストールを完了します。

macOS

1. 「Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for macOS」の手順を実行します。

   > **Important**
   >
   > 次のステップの下にある「ESP-IDF の取得」の手順に到達したらそこで止まり、このページの手順に戻ります。

2. コマンドラインウィンドウを開きます。
3. FreeRTOS ダウンロードディレクトリに移動し、次のスクリプトを実行して、お使いのプラットフォーム用の espressif ツールチェーンをダウンロードしてインストールします。

   ```
   vendors/espressif/esp-idf/install.sh
   ```

4. 次のコマンドを使用して、ESP-IDF ツールチェーンツールをターミナルのパスに追加します。

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

Linux

1. 「Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for Linux」の手順を実行します。

   > **Important**
   >
   > 次のステップの下にある「ESP-IDF の取得」の手順に到達したらそこで止まり、このページの手順に戻ります。

2. コマンドラインウィンドウを開きます。
3. FreeRTOS ダウンロードディレクトリに移動し、次のスクリプトを実行して、お使いのプラットフォーム用の espressif ツールチェーンをダウンロードしてインストールします。

   ```
   vendors/espressif/esp-idf/install.sh
   ```

4. 次のコマンドを使用して、ESP-IDF ツールチェーンツールをターミナルのパスに追加します。

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

4. シリアル接続を確立します。

   a. ホストマシンと ESP32-WROOM-32SE の間にシリアル接続を確立するには、CP210x USB を UART Bridge VCP ドライバーにインストールする必要があります。これらのドライバーは Silicon Labs からダウンロードできます。

   b. ESP32 でシリアル接続を確立する手順に従います。

   c. シリアル接続を確立したら、ボードとの接続用のシリアルポートをメモしておきます。デモをフラッシュするにはこれが必要です。

FreeRTOS デモアプリケーションを設定する

このチュートリアルでは、FreeRTOS 設定ファイルは *freertos*/vendors/espressif/boards/*board-name*/aws_demos/config_files/FreeRTOSConfig.h にあります。(例えば、AFR_BOARD espressif.esp32_devkitc を選択した場合、設定ファイルは *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/FreeRTOSConfig.h にあります。)

> **Important**
>
> ATECC608A デバイスには、プロジェクトが初めて実行されるとき (C_InitToken 呼び出し時) にデバイスにロックされる 1 回だけの初期化があります。ただし、FreeRTOS デモプロジェクトおよびテストプロジェクトには別の設定があります。デモプロジェクトの設定中にデバイスがロックされた場合、テストプロジェクトの一部のテストが成功しません。

1. FreeRTOS デモを設定する の手順に従って、FreeRTOS デモプロジェクトを設定します。最後のステップに進むと、 AWS IoT 認証情報をフォーマットするには、 を停止し、次の手順を実行します。
2. Microchip から、ATECC608A 部品のセットアップに役立ついくつかのスクリプトツールが提供されています。*freertos*/vendors/microchip/example_trust_chain_tool ディレクトリに移動し、README.md ファイルを開きます。
3. README.md ファイルの手順に従って、デバイスをプロビジョニングします。このステップには、以下が含まれます。
    1. 認証機関を作成して登録します AWS。
    2. ATECC608A でキーを生成し、パブリックキーとデバイスのシリアル番号をエクスポートする。
    3. デバイスの証明書を生成し、その証明書を に登録します AWS。
4. 開発者モードのキーのプロビジョニング の手順に従って、CA 認定とデバイス証明書をデバイスにロードします。

AWS クラウドでの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、次に [MQTT テストクライアント] を選択します。
3. [Subscription topic] (トピックのサブスクリプション) で *your-thing-name*/example/topic と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

idf.py スクリプトを使用して FreeRTOS デモプロジェクトを構築、フラッシュ、実行する

Espressif の IDF ユーティリティ (idf.py) を使用してビルドファイルを生成し、アプリケーションバイナリを構築し、バイナリをデバイスにフラッシュできます。

> **Note**
>
> 一部のセットアップでは、次の例のように、idf.py でポートオプション「-p port-name」を使用して正しいポートを指定する必要があります。
>
> ```
> idf.py -p /dev/cu.usbserial-00101301B flash
> ```

Windows、Linux、macOS で FreeRTOS を構築してフラッシュする (ESP-IDF v4.2)

1. FreeRTOS ダウンロードディレクトリのルートに移動します。
2. コマンドラインウィンドウで次のコマンドを入力して、ESP-IDF ツールをターミナルのパスに追加します。

   Windows (「コマンド」アプリ)

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Windows (「ESP-IDF 4.x CMD」アプリ)

   (これはアプリを開いた時点で既に完了しています。)

   Linux/macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```
3. 次のコマンドを使用して build ディレクトリで CMake を設定し、ファームウェアイメージを構築します。

   ```
   idf.py -DVENDOR=espressif -DBOARD=esp32_ecc608a_devkitc -DCOMPILER=xtensa-esp32
    build
   ```

   次の例のような出力が表示されます。

   ```
      Running cmake in directory /path/to/hello_world/build
   ```

```
   Executing "cmake -G Ninja --warn-uninitialized /path/to/hello_world"...
   Warn about uninitialized values.
   -- Found Git: /usr/bin/git (found version "2.17.0")
   -- Building empty aws_iot component due to configuration
   -- Component names: ...
   -- Component paths: ...

   ... (more lines of build system output)

   [527/527] Generating hello-world.bin
   esptool.py v2.3.1

   Project build complete. To flash, run this command:
   ../../../components/esptool_py/esptool/esptool.py -p (PORT) -b 921600
 write_flash --flash_mode dio --flash_size detect --flash_freq 40m 0x10000 build/
hello-world.bin  build 0x1000 build/bootloader/bootloader.bin 0x8000 build/
partition_table/partition-table.bin
   or run 'idf.py -p PORT flash'
```

エラーが起きなければ、構築によってファームウェアバイナリの .bin ファイルが生成されます。

4. 次のコマンドを使用して、開発ボードのフラッシュメモリを消去します。

```
idf.py erase_flash
```

5. idf.py スクリプトを使用して、アプリケーションバイナリをボードにフラッシュします。

```
idf.py flash
```

6. 次のコマンドを使用して、ボードのシリアルポートからの出力をモニタリングします。

```
idf.py monitor
```

> **Note**
>
> - 次の例のようにこれらのポリシーを組み合わせることができます。
>
>   ```
>   idf.py erase_flash flash monitor
>   ```

- 特定のホストマシンのセットアップでは、次の例のように、ボードをフラッシュするときにポートを指定する必要があります。

```
idf.py erase_flash flash monitor -p /dev/ttyUSB1
```

CMake で FreeRTOS を構築してフラッシュする

IDF SDK が提供する `idf.py` スクリプトを使ってコードを構築および実行できるほか、CMake を使用してプロジェクトを構築することもできます。現在、Unix Makefile と Ninja ビルドシステムをサポートしています。

プロジェクトを構築してフラッシュするには

1. コマンドラインウィンドウで、FreeRTOS ダウンロードディレクトリのルートに移動します。
2. 次のスクリプトを実行して、ESP-IDF ツールをシェルのパスに追加します。

   Windows

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Linux/macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

3. 次のコマンドを入力して、ビルドファイルを生成します。

   Unix Makefiles を使用

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32_plus_ecc608a_devkitc -DCOMPILER=xtensa-
   esp32 -S . -B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -
   DAFR_ENABLE_TESTS=0
   ```

   Ninja を使用

   ```
   cmake -DVENDOR=espressif -DBOARD=esp32_plus_ecc608a_devkitc -DCOMPILER=xtensa-
   esp32 -S . -B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -
   DAFR_ENABLE_TESTS=0 -GNinja
   ```

4. フラッシュを消去してから、ボードをフラッシュします。

   Unix Makefiles を使用

   ```
   make -C ./YOUR_BUILD_DIRECTORY erase_flash
   ```

   ```
   make -C ./YOUR_BUILD_DIRECTORY flash
   ```

   Ninja を使用

   ```
   ninja -C ./YOUR_BUILD_DIRECTORY erase_flash
   ```

   ```
   ninja -C ./YOUR_BUILD_DIRECTORY flash
   ```

追加情報

Espressif ESP32 ボードの使用とトラブルシューティングの詳細については、以下のトピックを参照してください。

- ESP32 用の独自の CMake プロジェクトで FreeRTOS を使用する
- トラブルシューティング
- デバッグ

Espressif ESP32-S2 の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

> **Note**
>
> 独自の Espressif IDF プロジェクト内で FreeRTOS モジュラーライブラリとデモを統合する方法の詳細については、ESP32-C3 プラットフォーム向けの注目のリファレンス統合を参照してください。

このチュートリアルでは、Espressif ESP32-S2 SoC と Esp32-s2-saola-1 開発ボードを開始する方法を示します。

概要

このチュートリアルでは次のステップを説明します。

1. ボードをホストマシンに接続します。
2. ホストマシンにソフトウェアをインストールし、マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
5. シリアル接続を使用して、実行中のアプリケーションを監視およびデバッグします。

前提条件

Espressif ボードで FreeRTOS の使用を開始する前に、 AWS アカウントとアクセス許可を設定する必要があります。

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一部では、電話またはテキストメッセージを受信し、電話のキーパッドに検証コードを入力します。

にサインアップすると AWS アカウント、 AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、ルートユーザーアクセスが必要なタスクの実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、 を保護し AWS IAM Identity Center、 を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、 AWS アカウント E メールアドレスを入力して、アカウント所有者AWS Management Consoleとして にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドのルートユーザーとしてサインインするを参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの AWS アカウント 「ルートユーザー (コンソール) の仮想 MFA デバイスを有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Centerの有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

を ID ソース IAM アイデンティティセンターディレクトリ として使用する方法のチュートリアルについては、「 AWS IAM Identity Center ユーザーガイド」の「デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「 ユーザーガイド」の AWS 「 アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの結合」を参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

  アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。
- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。
- IAM ユーザー:

- ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
- (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

はじめに

> **Note**
>
> このチュートリアルの Linux コマンドでは、Bash シェルを使用する必要があります。

1. Espressif ハードウェアを設定します。

   ESP32-S2 開発ボードハードウェアの設定の詳細については、ESP32-S2-Saola-1 入門ガイドを参照してください。

   > **Important**
   >
   > Espressif ガイドの開始方法セクションに到達したらそこで止まり、このページの手順に戻ります。

2. GitHub から Amazon FreeRTOS をダウンロードします。(手順については、README.md ファイルを参照してください。)
3. 開発環境をセットアップします。

   ボードと通信するには、ツールチェーンをインストールする必要があります。Espressif には、ボード用のソフトウェアを開発するための ESP-IDF が用意されています。ESP-IDF には独自のバージョンの FreeRTOS カーネルがコンポーネントとして統合されているため、Amazon FreeRTOS には、FreeRTOS カーネルが除去された、カスタムバージョンの ESP-IDF v4.2 が含まれています。これにより、コンパイル時にファイルが重複する問題が修正されます。Amazon FreeRTOS に含まれているカスタムバージョンの ESP-IDF v4.2 を使用するには、ホストマシンのオペレーティングシステムに応じて以下の手順を実行します。

   Windows

   1. ESP-IDF の Windows 用の汎用オンラインインストーラをダウンロードします。

2. 汎用オンラインインストーラを実行します。
3. ESP-IDF をダウンロードまたは使用する手順に進んだら、[既存の ESP-IDF ディレクトリを使用] を選択し、[既存の ESP-IDF ディレクトリを選択] を *freertos*/vendors/espressif/esp-idf に設定します。
4. インストールを完了します。

macOS

1. 「Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for macOS」の手順を実行します。

   > **Important**
   >
   > 次のステップの下にある「ESP-IDF の取得」の手順に到達したらそこで止まり、このページの手順に戻ります。

2. コマンドラインウィンドウを開きます。
3. FreeRTOS ダウンロードディレクトリに移動し、次のスクリプトを実行して、お使いのプラットフォーム用の espressif ツールチェーンをダウンロードしてインストールします。

   ```
   vendors/espressif/esp-idf/install.sh
   ```

4. 次のコマンドを使用して、ESP-IDF ツールチェーンツールをターミナルのパスに追加します。

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

Linux

1. 「Standard Setup of Toolchain prerequisites (ESP-IDF v4.2) for Linux」の手順を実行します。

   > **Important**
   >
   > 次のステップの下にある「ESP-IDF の取得」の手順に到達したらそこで止まり、このページの手順に戻ります。

2. コマンドラインウィンドウを開きます。
3. FreeRTOS ダウンロードディレクトリに移動し、次のスクリプトを実行して、お使いのプラットフォーム用の espressif ツールチェーンをダウンロードしてインストールします。

```
vendors/espressif/esp-idf/install.sh
```

4. 次のコマンドを使用して、ESP-IDF ツールチェーンツールをターミナルのパスに追加します。

```
source vendors/espressif/esp-idf/export.sh
```

4. シリアル接続を確立します。

   a. ホストマシンと ESP32-DevKitC の間にシリアル接続を確立するには、CP210x USB を UART Bridge VCP ドライバーにインストールします。これらのドライバーは Silicon Labs からダウンロードできます。

   b. ESP32 でシリアル接続を確立する手順に従います。

   c. シリアル接続を確立したら、ボードとの接続用のシリアルポートをメモしておきます。デモをフラッシュするにはこれが必要です。

FreeRTOS デモアプリケーションを設定する

このチュートリアルでは、FreeRTOS 設定ファイルは *freertos*/vendors/espressif/boards/*board-name*/aws_demos/config_files/FreeRTOSConfig.h にあります。(例えば、AFR_BOARD espressif.esp32_devkitc を選択した場合、設定ファイルは *freertos*/vendors/espressif/boards/esp32/aws_demos/config_files/FreeRTOSConfig.h にあります。)

1. macOS または Linux を実行している場合、ターミナルプロンプトを開きます。Windows を実行している場合は、「ESP-IDF 4.x CMD」アプリ (ESP-IDF ツールチェーンのインストール時にこのオプションを含めた場合) または「コマンドプロンプト」アプリ (オプションを含めなかった場合) を開きます。
2. Python3 がインストールされていることを確認するには、次のコマンドを実行します。

```
python --version
```

インストールされているバージョンが表示されます。Python 3.0.1 以降がインストールされていない場合は、Python ウェブサイトからインストールできます。

3. AWS IoT コマンドを実行するには AWS 、 コマンドラインインターフェイス (CLI) が必要です。Windows を実行している場合は、 `easy_install awscli` コマンドを使用して AWS 、「Command」または「ESP-IDF 4.x CMD」アプリに CLI をインストールします。

   macOS または Linux を実行している場合は、「 AWS CLI のインストール」を参照してください。

4. 実行

   ```
   aws configure
   ```

   アクセスキー ID AWS 、シークレット AWS アクセスキー、デフォルト AWS リージョンを使用して CLI を設定します。詳細については、AWS CLI の設定を参照してください。

5. 次のコマンドを使用して AWS SDK for Python (boto3) をインストールします。

   - Windows の場合、「コマンド」または「ESP-IDF 4.x CMD」アプリで、以下を実行します。

     ```
     easy_install boto3
     ```

   - macOS または Linux の場合、以下を実行します。

     ```
     pip install tornado nose --user
     ```

     続いて以下を実行します。

     ```
     pip install boto3 --user
     ```

FreeRTOS には、 AWS IoTに接続するための Espressif ボードのセットアップを容易にする `SetupAWS.py` スクリプトが含まれています。

設定スクリプトを実行するには

1. このスクリプトを設定するには、*freertos*`/tools/aws_config_quick_start/configure.json` を開いて以下の属性を設定します。

afr_source_dir

コンピュータの *freertos* ディレクトリへの完全なパス。このパスの指定にスラッシュを使用していることを確認します。

thing_name

ボードを表す AWS IoT モノに割り当てる名前。

wifi_ssid

Wi-Fi ネットワークの SSID。

wifi_password

Wi-Fi ネットワークのパスワード。

wifi_security

Wi-Fi ネットワークのセキュリティタイプ。次のセキュリティタイプが有効です。

- eWiFiSecurityOpen (オープン、セキュリティなし)
- eWiFiSecurityWEP (WEP セキュリティ)
- eWiFiSecurityWPA (WPA セキュリティ)
- eWiFiSecurityWPA2 (WPA2 セキュリティ)

2. macOS または Linux を実行している場合、ターミナルプロンプトを開きます。Windows を実行している場合は、「ESP-IDF 4.x CMD」アプリまたは「コマンド」アプリを開きます。
3. *freertos*/tools/aws_config_quick_start ディレクトリに移動して、以下を実行します。

```
python SetupAWS.py setup
```

スクリプトは、次を実行します。

- AWS IoT モノ、証明書、ポリシーを作成します。
- AWS IoT ポリシーを証明書にアタッチし、証明書を AWS IoT モノにアタッチします。
- aws_clientcredential.h エンドポイント、Wi-Fi SSID、および認証情報を AWS IoT ファイルに追加します。
- 証明書とプライベートキーをフォーマットして aws_clientcredential_keys.h ヘッダーファイルに書き込みます。

> **Note**
>
> 証明書はデモ目的でのみハードコードされています。本番稼動レベルのアプリケーションでは、これらのファイルを安全な場所に保存する必要があります。

SetupAWS.py の詳細については、*freertos*/tools/aws_config_quick_start ディレクトリにある「README.md」を参照してください。

AWS クラウドでの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、次に [MQTT テストクライアント] を選択します。
3. [Subscription topic] (トピックのサブスクリプション) で *your-thing-name*/example/topic と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

idf.py スクリプトを使用して FreeRTOS デモプロジェクトを構築、フラッシュ、実行する

Espressif の IDF ユーティリティを使用してビルドファイルを生成し、アプリケーションバイナリを構築し、ボードをフラッシュできます。

Windows、Linux、macOS で FreeRTOS を構築してフラッシュする (ESP-IDF v4.2)

idf.py スクリプトを使用して、プロジェクトを構築し、デバイスにバイナリをフラッシュします。

> **Note**
>
> 一部のセットアップでは、次の例のように、idf.py でポートオプション -p port-name を使用して正しいポートを指定する必要があります。

```
idf.py -p /dev/cu.usbserial-00101301B flash
```

プロジェクトを構築してフラッシュするには

1. FreeRTOS ダウンロードディレクトリのルートに移動します。
2. コマンドラインウィンドウで次のコマンドを入力して、ESP-IDF ツールをターミナルのパスに追加します。

   Windows (「コマンド」アプリ)

   ```
   vendors\espressif\esp-idf\export.bat
   ```

   Windows (「ESP-IDF 4.x CMD」アプリ)

   (これはアプリを開いた時点で既に完了しています。)

   Linux/macOS

   ```
   source vendors/espressif/esp-idf/export.sh
   ```

3. 次のコマンドを使用して build ディレクトリで CMake を設定し、ファームウェアイメージを構築します。

   ```
   idf.py -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 build
   ```

   次の例のような出力が表示されます。

   ```
   Executing action: all (aliases: build)
      Running cmake in directory /path/to/hello_world/build
      Executing "cmake -G Ninja -DPYTHON_DEPS_CHECKED=1 -DESP_PLATFORM=1
    -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 -
   DCCACHE_ENABLE=0 /path/to/hello_world"...
      -- The C compiler identification is GNU 8.4.0
      -- The CXX compiler identification is GNU 8.4.0
      -- The ASM compiler identification is GNU

      ... (more lines of build system output)
   ```

```
  [1628/1628] Generating binary image from built executable
  esptool.py v3.0
  Generated /path/to/hello_world/build/aws_demos.bin

  Project build complete. To flash, run this command:
  esptool.py -p (PORT) -b 460800 --before default_reset --after hard_reset --chip
esp32s2  write_flash --flash_mode dio --flash_size detect --flash_freq 80m 0x1000
build/bootloader/bootloader.bin 0x8000 build/partition_table/partition-table.bin
0x16000 build/ota_data_initial.bin 0x20000 build/aws_demos.bin
  or run 'idf.py -p (PORT) flash'
```

エラーが発生していない場合、構築によってファームウェアバイナリの .bin ファイルが生成されます。

4. 次のコマンドを使用して、開発ボードのフラッシュメモリを消去します。

```
idf.py erase_flash
```

5. idf.py スクリプトを使用して、アプリケーションバイナリをボードにフラッシュします。

```
idf.py flash
```

6. 次のコマンドを使用して、ボードのシリアルポートからの出力をモニタリングします。

```
idf.py monitor
```

> **Note**
>
> - 次の例のようにこれらのポリシーを組み合わせることができます。
>
>   ```
>   idf.py erase_flash flash monitor
>   ```
>
> - 特定のホストマシンのセットアップでは、次の例のように、ボードをフラッシュするときにポートを指定する必要があります。
>
>   ```
>   idf.py erase_flash flash monitor -p /dev/ttyUSB1
>   ```

CMake で FreeRTOS を構築してフラッシュする

IDF SDK が提供する `idf.py` スクリプトを使ってコードを構築および実行できるほか、CMake を使用してプロジェクトを構築することもできます。現在、Unix Makefile と Ninja ビルドシステムをサポートしています。

プロジェクトを構築してフラッシュするには

1. コマンドラインウィンドウで、FreeRTOS ダウンロードディレクトリのルートに移動します。
2. 次のスクリプトを実行して、ESP-IDF ツールをシェルのパスに追加します。

   - Windows

     ```
     vendors\espressif\esp-idf\export.bat
     ```

   - Linux/macOS

     ```
     source vendors/espressif/esp-idf/export.sh
     ```

3. 次のコマンドを入力して、ビルドファイルを生成します。

   - Unix Makefiles を使用

     ```
     cmake -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 -S . -
     B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0
     ```

   - Ninja を使用

     ```
     cmake -DVENDOR=espressif -DBOARD=esp32s2_saola_1 -DCOMPILER=xtensa-esp32s2 -S . -
     B ./YOUR_BUILD_DIRECTORY -DAFR_ENABLE_ALL_MODULES=1 -DAFR_ENABLE_TESTS=0 -GNinja
     ```

4. プロジェクトをビルドします。

   - Unix Makefiles を使用

     ```
     make -C ./YOUR_BUILD_DIRECTORY -j8
     ```

   - Ninja を使用

     ```
     ninja -C ./YOUR_BUILD_DIRECTORY -j8
     ```

5. フラッシュを消去してから、ボードをフラッシュします。

- Unix Makefiles を使用

```
make -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
make -C ./YOUR_BUILD_DIRECTORY flash
```

- Ninja を使用

```
ninja -C ./YOUR_BUILD_DIRECTORY erase_flash
```

```
ninja -C ./YOUR_BUILD_DIRECTORY flash
```

追加情報

Espressif ESP32 ボードの使用とトラブルシューティングの詳細については、以下のトピックを参照してください。

- ESP32 用の独自の CMake プロジェクトで FreeRTOS を使用する
- トラブルシューティング
- デバッグ

Infineon XMC4800 IoT 接続キットの開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Infineon XMC4800 IoT 接続キットの使用を開始するための手順について説明します。Infineon XMC4800 IoT Connectivity Kit がない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

ボードとのシリアル接続を開いてログ記録とデバッグ情報を表示するには、XMC4800 IoT 接続キットに加えて、3.3V USB/シリアルコンバータが必要です。CP2104 は、Adafruit の CP2104 Friend などのボードで広く入手できる、一般的な USB/シリアルコンバータです。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
4. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

開発環境をセットアップする

FreeRTOS は、XMC4800 のプログラミングに Infineon の DAVE 開発環境を使用します。開始する前に、DAVE および 一部の J-Link ドライバーをダウンロードしてインストールする必要があります。これにより、オンボードデバッガーと通信できるようになります。

DAVE をインストールする

1. Infineon の DAVE software download ページに移動します。
2. ご使用のオペレーティングシステム向けの DAVE パッケージを選択し、登録情報を送信します。Infineon での登録後、.zip ファイルをダウンロードするためのリンクが記載された確認メールを受け取ります。
3. DAVE パッケージ .zip ファイル (`DAVE_`*`version_os_date`*`.zip`) をダウンロードして、DAVE をインストールする場所 (例えば、`C:\DAVE4`) に解凍します。

> **Note**
>
> 一部の Windows ユーザーから、ファイルの解凍に Windows エクスプローラーを使用する際の問題が報告されています。7-Zip などのサードパーティー製のプログラムを使用することをお勧めします。

4. DAVE を起動するには、解凍された DAVE_*version_os_date*.zip フォルダで検出された実行可能ファイルを実行します。

詳細については DAVE Quick Start Guide を参照してください。

Segger J-Link ドライバーをインストールする

XMC4800 Relax EtherCAT ボードのオンボードデバッグプローブと通信するには、J-Link ソフトウェアおよびドキュメントパックに含まれているドライバが必要です。Segger の J-Link software download ページから、J-Link ソフトウェアとドキュメントパックをダウンロードできます。

シリアル接続の確立

シリアル接続の設定はオプションですが、推奨されています。シリアル接続により、開発マシンで表示可能な形式でボードからログとデバッグ情報を送信できるようになります。

XMC4800 デモアプリケーションは、ピン P0.0 および P0.1 の UART シリアル接続を使用します。このピンは、XMC4800 Relax EtherCAT ボードのシルクスクリーンにラベル表示されています。シリアル接続の設定方法は、以下の通りです。

1. 「RX<P0.0」とラベル付けされたピンを、お使いの USB/シリアルコンバータの「TX」ピンに接続します。
2. 「TX>P0.1」とラベル付けされたピンを、お使いの USB/シリアルコンバータの「RX」ピンに接続します。
3. シリアルコンバータの Ground (接地) ピンを、ボード上の「GND」とラベル付けされたピンのいずれかに接続します。デバイスは、共通の接地を持っている必要があります。

電力は USB デバッグポートから供給されるため、シリアルアダプタの正電圧ピンをボードに接続することはしないでください。

> **Note**
>
> 一部のシリアルケーブルは、5V シグナルレベルを使用します。XMC4800 ボードと Wi-Fi クリックモジュールには、3.3V が必要です。ボードのシグナルを 5V に変更するために、ボードの IOREF ジャンパーを使用することはしないでください。

ケーブルが接続されると、GNU Screen などのターミナルエミュレーターでシリアル接続を開くことができます。ボーレートはデフォルトで 115200 に設定されており、8 データビット、パリティなし、1 ストップビットです。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモを DAVE にインポートする

1. DAVE を起動します。
2. DAVE で、[File] (ファイル)、[Import] (インポート) の順に選択します。[Import] (インポート) ウィンドウで、[Infineon] フォルダを展開し、[DAVE Project] (DAVE プロジェクト) を選択してから [Next] (次へ) を選択します。
3. [Import DAVE Projects] (DAVE プロジェクトのインポート) で、[Select Root Directory] (ルートディレクトリの選択)、[Browse] (参照) の順に選択してから、XMC4800 デモプロジェクトを選択します。

FreeRTOS のダウンロードを解凍したディレクトリで、デモプロジェクトは `projects/infineon/xmc4800_iotkit/dave4/aws_demos` にあります。

[Copy Projects Into Workspace] (プロジェクトをワークスペースにコピー) がオフになっていることを確認します。

4. [Finish] (終了) を選択します。

   `aws_demos` プロジェクトは、WorkSpace にインポートされ、アクティブ化されます。

5. [Project] (プロジェクト) メニューから [Build Active Project] (アクティブなプロジェクトを構築) を選択します。

   プロジェクトがエラーなしでビルドされていることを確認します。

FreeRTOS デモプロジェクトを実行する

1. USB ケーブルを使用して XMC4800 IoT 接続キットをコンピュータに接続します。このボードには 2 つの microUSB コネクタがあります。「X101」とラベル付けされたものを使用します。ボードのシルクスクリーン上で、デバッグがその横に表示されています。
2. [Project] (プロジェクト) メニューから、[Rebuild Active Project] (アクティブなプロジェクトの再構築) を選択して、`aws_demos` を再構築し、設定変更が取得されたことを確認します。
3. [Project Explorer] (プロジェクトエクスプローラー) から `aws_demos` を右クリックして [Debug As] (デバッグ方法) を選択し、[DAVE C/C++ Application] (DAVE C/C++ アプリケーション) を選択します。
4. [GDB SEGGER J-Link Debugging] (GDB SEGGER J-Link デバッグ) をダブルクリックして、デバッグ情報を作成します。[Debug] (デバッグ) を選択します。
5. デバッガーが `main( )` のブレークポイントで停止したら、[Run] (実行) メニューから [Resume] (再開) を選択します。

AWS IoT コンソールでは、ステップ 4 ~ 5 の MQTT クライアントに、デバイスから送信された MQTT メッセージが表示されます。シリアル接続を使用する場合は、UART 出力で次のように表示されます。

```
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----
```

CMake で FreeRTOS デモを構築する

FreeRTOS 開発に IDE を使用しない場合は、代わりに CMake を使用して、サードパーティのコードエディタおよびデバッグツールを使用して開発したデモアプリケーションまたはアプリケーションを構築して実行できます。

> **Note**
>
> このセクションでは、ネイティブビルドシステムとして MinGW を使用した Windows で CMake を使用する方法について説明します。他のオペレーティングシステムおよびオプションで CMake を使用する方法の詳細については、「FreeRTOS で CMake を使用する」を参照してください。(MinGW は、ネイティブ Microsoft Windows アプリケーション用の最小限の開発環境です。)

CMake で FreeRTOS デモを構築するには

1. GNU Arm Embedded Toolchain をセットアップします。

   a. Arm Embedded Toolchain ダウンロードページから Windows バージョンのツールチェーンをダウンロードします。

   > **Note**
   >
   > 「8-2018-q4-major」のバージョンの「objcopy」ユーティリティでバグが報告されているため、「8-2018-q4-major」以外のバージョンをダウンロードすることをお勧めします。

   b. ダウンロードしたツールチェーンインストーラーを開き、インストーラインストールウィザードの手順に従ってツールチェーンをインストールします。

   > **Important**
   >
   > インストールウィザードの最後のページで、[Add path to environment variable] (パスを環境変数に追加) を選択してツールチェーンパスをシステムのパス環境変数に追加します。

2. CMake および MinGW をインストールします。

   手順については、「CMake の前提条件」を参照してください。

3. 生成されたビルドファイルを格納するフォルダ (*build-folder*) を作成します。

4. ディレクトリを FreeRTOS ダウンロードディレクトリ (*freertos*) に変更し、次のコマンドを使用してビルドファイルを生成します。

```
cmake -DVENDOR=infineon -DBOARD=xmc4800_iotkit -DCOMPILER=arm-gcc -S . -B build-folder -G "MinGW Makefiles" -DAFR_ENABLE_TESTS=0
```

5. ディレクトリをビルドディレクトリ (*build-folder*) に変更し、次のコマンドを使用してバイナリをビルドします。

   ```
   cmake --build . --parallel 8
   ```

   このコマンドにより、ビルドディレクトリにバイナリ出力 aws_demos.hex がビルドされます。

6. JLINK を使用してイメージをフラッシュして実行します。

   a. ビルドディレクトリ (*build-folder*) から、次のコマンドを使用してフラッシュスクリプトを作成します。

      ```
      echo loadfile aws_demos.hex > flash.jlink
      ```

      ```
      echo r >> flash.jlink
      ```

      ```
      echo g >> flash.jlink
      ```

      ```
      echo q >> flash.jlink
      ```

   b. JLNIK 実行可能ファイを使用してイメージをフラッシュします。

      ```
      JLINK_PATH\JLink.exe  -device XMC4800-2048 -if SWD -speed auto -CommanderScript
       flash.jlink
      ```

      ボードで確立したシリアル接続を介してアプリケーションログが表示されます。

トラブルシューティング

デバイスを AWS クラウドに接続するために、 AWS IoT と FreeRTOS ダウンロードをまだ設定していない場合は、必ず設定してください。手順については「最初のステップ」を参照してください。

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Infineon OPTIGA Trust X と XMC4800 IoT Connectivity Kit の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Infineon OPTIGA Trust X セキュアエレメントと XMC4800 IoT Connectivity Kit の使用を開始するための手順について説明します。Infineon XMC4800 IoT 接続キットの開始方法 チュートリアルとは異なり、このガイドでは、Infineon OPTIGA Trust X セキュアエレメントを使用してセキュアな認証情報を提供する方法を示します。

以下のハードウェアが必要です。

1. ホスト MCU - Infineon XMC4800 IoT Connectivity Kit については、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。
2. セキュリティ拡張パック:
   - セキュアエレメント - Infineon OPTIGA Trust X。

     AWS Partner Device Catalog にアクセスして、当社のパートナーから購入してください。
   - パーソナライゼーションボード - Infineon OPTIGA パーソナライゼーションボード。
   - アダプターボード - Infineon MyIoT アダプター。

ここで説明する手順を実行するには、ボードとのシリアル接続を開いて、ログおよびデバッグ情報を表示する必要があります (いずれかのステップで、ボードのシリアルデバッグ出力からパブリックキーをコピーし、ファイルに貼り付ける必要があります)。そのためには、XMC4800 IoT Connectivity Kit に加えて、3.3V USB/シリアルコンバータが必要です。JBtek EL-PN-47310126 USB/シリアルコンバータは、このデモで動作することがわかっています。シリアルケーブルを Infineon MyIoT アダプターボードに接続するには、3 本のオス - オスジャンパー線 (受信 (RX)、送信 (TX)、接地 (GND) 用) も必要です。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については、オプション #2: オンボードプライベートキーの生成 を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。

概要

このチュートリアルでは、以下の手順が含まれています。

1. ホストマシンにソフトウェアをインストールし、マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
4. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションを操作します。

開発環境をセットアップする

FreeRTOS は、XMC4800 のプログラミングに Infineon の DAVE 開発環境を使用します。開始する前に、DAVE および一部の J-Link ドライバーをダウンロードしてインストールし、オンボードデバッガーと通信できるようにします。

DAVE をインストールする

1. Infineon の DAVE software download ページに移動します。
2. ご使用のオペレーティングシステム向けの DAVE パッケージを選択し、登録情報を送信します。登録後、.zip ファイルをダウンロードするためのリンクが記載された確認メールが届きます。
3. DAVE パッケージ .zip ファイル (`DAVE_version_os_date.zip`) をダウンロードして、DAVE をインストールする場所 (例えば、`C:\DAVE4`) に解凍します。

   > **Note**
   >
   > 一部の Windows ユーザーから、ファイルの解凍に Windows エクスプローラーを使用する際の問題が報告されています。7-Zip などのサードパーティー製のプログラムを使用することをお勧めします。

4. DAVE を起動するには、解凍された `DAVE_version_os_date.zip` フォルダで検出された実行可能ファイルを実行します。

詳細については DAVE Quick Start Guide を参照してください。

Segger J-Link ドライバーをインストールする

XMC4800 IoT Connectivity Kit のオンボードデバッグプローブと通信するには、J-Link ソフトウェアおよびドキュメントパックに含まれているドライバが必要です。Segger の J-Link software download ページから、J-Link ソフトウェアとドキュメントパックをダウンロードできます。

シリアル接続の確立

USB/シリアルコンバータケーブルを Infineon Shield2Go Adapter に接続します。これにより、開発マシンで表示可能な形式でボードからログとデバッグ情報を送信できるようになります。シリアル接続の設定方法は、以下の通りです。

1. RX ピンを USB/シリアルコンバータの TX ピンに接続します。
2. TX ピンを USB/シリアルコンバータの RX ピンに接続します。
3. シリアルコンバータの Ground (接地) ピンを、ボード上の GND ピンのいずれかに接続します。デバイスは、共通の接地を持っている必要があります。

電力は USB デバッグポートから供給されるため、シリアルアダプタの正電圧ピンをボードに接続することはしないでください。

> **Note**
>
> 一部のシリアルケーブルは、5V シグナルレベルを使用します。XMC4800 ボードと Wi-Fi クリックモジュールには、3.3V が必要です。ボードのシグナルを 5V に変更するために、ボードの IOREF ジャンパーを使用することはしないでください。

ケーブルが接続されると、GNU Screen などのターミナルエミュレーターでシリアル接続を開くことができます。ボーレートはデフォルトで 115200 に設定されており、8 データビット、パリティなし、1 ストップビットです。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。

2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモを DAVE にインポートする

1. DAVE を起動します。
2. DAVE で、[File] (ファイル) を選択し、[Import] (インポート) を選択します。[Infineon] フォルダを展開し、[DAVE Project] (DAVE プロジェクト) を選択してから [Next] (次へ) を選択します。
3. [Import DAVE Projects] (DAVE プロジェクトのインポート) で、[Select Root Directory] (ルートディレクトリの選択)、[Browse] (参照) の順に選択してから、XMC4800 デモプロジェクトを選択します。

   FreeRTOS のダウンロードを解凍したディレクトリで、デモプロジェクトは `projects/infineon/xmc4800_plus_optiga_trust_x/dave4/aws_demos/dave4` にあります。

   [Copy Projects Into Workspace] (プロジェクトを Workspace にコピー) がオフになっていることを確認します。
4. [Finish] (終了) を選択します。

   `aws_demos` プロジェクトは、WorkSpace にインポートされ、アクティブ化されます。
5. [Project] (プロジェクト) メニューから [Build Active Project] (アクティブなプロジェクトを構築) を選択します。

   プロジェクトがエラーなしでビルドされていることを確認します。

FreeRTOS デモプロジェクトを実行する

1. [Project] (プロジェクト) メニューから、[Rebuild Active Project] (アクティブなプロジェクトの再構築) を選択して、`aws_demos` を再構築し、設定変更が取得されたことを確認します。

2. [Project Explorer] (プロジェクトエクスプローラー) から aws_demos を右クリックして [Debug As] (デバッグ方法) を選択し、[DAVE C/C++ Application] (DAVE C/C++ アプリケーション) を選択します。
3. [GDB SEGGER J-Link Debugging] (GDB SEGGER J-Link デバッグ) をダブルクリックして、デバッグ情報を作成します。[Debug] (デバッグ) を選択します。
4. デバッガーが main( ) のブレークポイントで停止したら、[Run] (実行) メニューから [Resume] (再開) を選択します。

この時点で、オプション #2: オンボードプライベートキーの生成 のパブリックキーの抽出手順に進みます。すべてのステップが完了したら、 AWS IoT コンソールに移動します。以前に設定した MQTT クライアントは、デバイスから送信された MQTT メッセージを表示します。デバイスのシリアル接続を介して UART 出力が以下のように表示されます。

```
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----
```

CMake で FreeRTOS デモを構築する

このセクションでは、ネイティブビルドシステムとして MinGW を使用した Windows で CMake を使用する方法について説明します。他のオペレーティングシステムおよびオプションで CMake を使用する方法の詳細については、「FreeRTOS で CMake を使用する」を参照してください。(MinGW は、ネイティブ Microsoft Windows アプリケーション用の最小限の開発環境です。)

FreeRTOS 開発に IDE を使用しない場合は、代わりに CMake を使用して、サードパーティーのコードエディタおよびデバッグツールを使用して開発した、デモアプリケーションまたはアプリケーションを構築して実行できます。

CMake で FreeRTOS デモを構築するには

1. GNU Arm Embedded Toolchain をセットアップします。

   a. Arm Embedded Toolchain ダウンロードページから Windows バージョンのツールチェーンをダウンロードします。

      > **Note**
      >
      > 「8-2018-q4-major」バージョンについては、objcopy ユーティリティでバグが報告されているため、それ以外のバージョンをダウンロードすることをお勧めします。

   b. ダウンロードしたツールチェーンインストーラーを開き、ウィザードの指示に従います。

   c. インストールウィザードの最後のページで、[Add path to environment variable] (パスを環境変数に追加) を選択してツールチェーンパスをシステムのパス環境変数に追加します。

2. CMake および MinGW をインストールします。

手順については、「CMake の前提条件」を参照してください。

3. 生成されたビルドファイルを格納するフォルダ (*build-folder*) を作成します。
4. ディレクトリを FreeRTOS ダウンロードディレクトリ (*freertos*) に変更し、次のコマンドを使用してビルドファイルを生成します。

```
cmake -DVENDOR=infineon -DBOARD=xmc4800_plus_optiga_trust_x -DCOMPILER=arm-gcc -S .
 -B build-folder -G "MinGW Makefiles" -DAFR_ENABLE_TESTS=0
```

5. ディレクトリをビルドディレクトリ (*build-folder*) に変更し、次のコマンドを使用してバイナリをビルドします。

```
cmake --build . --parallel 8
```

このコマンドにより、ビルドディレクトリにバイナリ出力 aws_demos.hex がビルドされます。

6. JLINK を使用してイメージをフラッシュして実行します。

   a. ビルドディレクトリ (*build-folder*) から、次のコマンドを使用してフラッシュスクリプトを作成します。

```
echo loadfile aws_demos.hex > flash.jlink
echo r >> flash.jlink
echo g >> flash.jlink
echo q >> flash.jlink
```

   b. JLNIK 実行可能ファイを使用してイメージをフラッシュします。

```
JLINK_PATH\JLink.exe  -device XMC4800-2048 -if SWD -speed auto -CommanderScript
 flash.jlink
```

ボードで確立したシリアル接続を介してアプリケーションログが表示されます。続いて、オプション #2: オンボードプライベートキーの生成 のパブリックキーの抽出手順に進みます。すべてのステップが完了したら、 AWS IoT コンソールに移動します。以前に設定した MQTT クライアントは、デバイスから送信された MQTT メッセージを表示します。

トラブルシューティング

一般的なトラブルシューティング情報については、「トラブルシューティングの開始方法」を参照してください。

MW32x AWS IoT スターターキットの開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

AWS IoT スターターキットは、NXP の最新の統合 Cortex M4 マイクロコントローラーである 88MW320/88MW322 に基づく開発キットで、802.11b/g/n Wi-Fi を単一のマイクロコントローラーチップに統合しています。FCC 認定されている開発キットです。パートナーから購入する方法の詳細については、AWS パートナーデバイスカタログを参照してください。また、88MW320/88MW322 モジュールも FCC 認定されており、カスタマイズと大量販売が可能です。

この入門ガイドでは、ホストコンピュータで SDK とアプリケーションをクロスコンパイルし、SDK で提供されているツールを使用して、生成されたバイナリファイルをボードにロードする方法を説明します。アプリケーションがボード上で稼働を開始すると、ホストコンピュータのシリアルコンソールからデバッグまたは操作できます。

開発とデバッグでサポートされているホストプラットフォームは Ubuntu 16.04 です。他のプラットフォームも使用できる可能性はありますが、公式にはサポートされていません。ホストプラットフォームにソフトウェアをインストールするには、アクセス許可が必要です。SDK を構築するには、次の外部ツールが必要です。

- Ubuntu 16.04 ホストプラットフォーム
- ARM ツールチェーンバージョン 4_9_2015q3
- Eclipse 4.9.0 IDE

ARM ツールチェーンは、アプリケーションと SDK をクロスコンパイルするために必要です。SDK はツールチェーンの最新バージョンを利用して、イメージフットプリントを最適化し、より多くの機能を少ないスペースに収めます。このガイドでは、ツールチェーンのバージョン 4_9_2015q3 を使

用していることを前提としています。古いバージョンのツールチェーンを使用することはお勧めしません。開発キットは、ワイヤレスマイクロコントローラーのデモプロジェクトのファームウェアで事前にフラッシュされています。

トピック

- ハードウェアの設定
- 開発環境のセットアップ
- FreeRTOS デモプロジェクトを構築して実行する
- デバッグ
- トラブルシューティング

ハードウェアの設定

ミニ USB や USB ケーブルを使用して、MW32x ボードをラップトップに接続します。ミニ USB ケーブルは、ボード上の唯一のミニ USB コネクタに接続します。ジャンパー交換は不要です。

ボードがラップトップまたはデスクトップコンピュータに接続されている場合は、外部電源は必要ありません。

この USB 接続では、次の機能が提供されます。

- ボードへのコンソールアクセス。仮想 tty/com ポートは、コンソールへのアクセスに使用できる開発ホストに登録されます。
- JTAG のボードへのアクセス。これは、ボードの RAM またはフラッシュにファームウェアイメージをロードまたはアンロードしたり、デバッグしたりするために使用できます。

開発環境のセットアップ

開発目的での最小要件は、ARM ツールチェーンと SDK にバンドルされているツールです。次のセクションでは、ARM ツールチェーンの設定について説明します。

GNU ツールチェーン

SDK は GCC コンパイラツールチェーンを正式にサポートしています。GNU ARM のクロスコンパイラツールチェーンは、GNU Arm Embedded Toolchain 4.9-2015-q3-update にあります。

ビルドシステムは、デフォルトで GNU ツールチェーンを使用するように設定されています。Makefiles は、GNU コンパイラツールチェーンバイナリがユーザーのパスで利用可能であ

り、Makefiles から呼び出すことができることを前提としています。また、Makefiles は GNU ツールチェーンバイナリのファイル名の先頭に `arm-none-eabi-` が付いていることを想定しています。

GCC ツールチェーンを GDB と共に使用して OpenOCD (SDK にバンドルされている) でデバッグできます。これにより、JTAG と連携するソフトウェアが提供されます。

gcc-arm-embedded ツールチェーンのバージョン 4_9_2015q3 をお勧めします。

Linux ツールチェーンのセットアップ手順

Linux で GCC ツールチェーンを設定するには、以下の手順に従います。

1. ツールチェーンの tarball は、GNU Arm Embedded Toolchain 4.9-2015-q3-update からダウンロードできます。ファイルは `gcc-arm-none-eabi-4_9-2015q3-20150921-linux.tar.bz2` です。
2. このファイルを任意のディレクトリにコピーします。ディレクトリ名にスペースが含まれていないことを確認してください。
3. 次のコマンドを使用してファイルを展開します。

   ```
   tar -vxf filename
   ```

4. インストールされたツールチェーンのパスをシステムのパスに追加します。例えば、`/home/user-name` ディレクトリにある `.profile` ファイルの末尾に次の行を追加します。

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

> **Note**
>
> 新しい Ubuntu のディストリビューションには、GCC クロスコンパイラの Debian バージョンが含まれる場合があります。その場合、ネイティブのクロスコンパイラを削除し、上記のセットアップ手順を行う必要があります。

Linux 開発ホストでの作業

Ubuntu や Fedora などの最新の Linux デスクトップディストリビューションはどれでも使用できます。ただし、最新リリースにアップグレードすることをお勧めします。次の手順は、Ubuntu 16.04 で動作することが確認されており、そのバージョンを使用していることを前提としています。

パッケージのインストール

SDK には、新しくセットアップされた Linux マシンでの開発環境の迅速なセットアップを可能にするスクリプトが含まれています。スクリプトは、マシンタイプを自動的に検出し、C ライブラリ、USB ライブラリ、FTDI ライブラリ、ncurses、Python、LaTeX などの適切なソフトウェアをインストールしようとします。このセクションでは、汎用ディレクトリ名は AWS SDK ルートディレクトリ*amzsdk_bundle-x.y.z*を示します。実際のディレクトリ名は異なる場合があります。ルート権限が必要です。

- *amzsdk_bundle-x.y.z/* ディレクトリに移動し、このコマンドを実行します。

```
./lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/bin/installpkgs.sh
```

sudo の回避

以下の説明のとおり、このガイドでは flashprog オペレーションで flashprog.py スクリプトを使用して、ボードの NAND をフラッシュします。同様に、ramload オペレーションでは ramload.py スクリプトを使用して、NAND をフラッシュせずにホストからマイクロコントローラーの RAM にファームウェアイメージを直接コピーします。

Linux 開発ホストを設定して、sudo コマンドを毎回必要とせずに flashprog および ramload オペレーションを実行できるようになります。これを行うには、以下のコマンドを実行します。

```
./lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/bin/perm_fix.sh
```

> **Note**
>
> スムーズな Eclipse IDE エクスペリエンスを実現するように、Linux 開発ホストの権限を設定する必要があります。

シリアルコンソールの設定

USB ケーブルを Linux ホスト USB スロットに挿入します。これにより、デバイスの検出がトリガーされます。/var/log/messages ファイル、または dmesg コマンド実行後に次のようなメッセージが表示されます。

```
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
```

2 つの ttyUSB デバイスが作成済みであることを確認します。2 つ目の ttyUSB はシリアルコンソールです。上記の例では「ttyUSB1」という名前です。

このガイドでは、minicom を使用してシリアルコンソールの出力を確認します。また、putty のような他のシリアルプログラムを使用することもできます。以下のコマンドを実行して、minicom をセットアップモードで実行します。

```
minicom -s
```

minicom で、[Serial Port Setup] (シリアルポートの設定) に移動し、次の設定をキャプチャします。

```
| 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
```

今後使用できるように、これらの設定を minicom に保存することができます。シリアルコンソールからのメッセージが minicom ウィンドウに表示されます。

シリアルコンソールウィンドウを選択し、Enter キーを押します。画面にハッシュ (#) が表示されます。

> **Note**
>
> 開発ボードには、FTDI シリコンデバイスが含まれています。FTDI デバイスは、ホストに 2 つの USB インターフェイスを提供します。最初のインターフェイスは MCU の JTAG 機能に関連付けられ、2 つ目のインターフェイスは MCU の物理 UARTx ポートに関連付けられます。

OpenOCD のインストール

OpenOCD は、組み込みターゲットデバイスのデバッグ、システム内プログラミング、バウンダリスキャンテストを提供するソフトウェアです。

OpenOCD バージョン 0.9 が必要です。Eclipse の機能にも必要です。バージョン 0.7 などの以前のバージョンが Linux ホストにインストールされている場合は、現在使用している Linux ディストリビューションに適したコマンドを使用して、そのリポジトリを削除します。

標準の Linux コマンドを実行して OpenOCD をインストールします。

```
apt-get install openocd
```

上記のコマンドでバージョン 0.9 以降がインストールされない場合は、次の手順を使用して openocd ソースコードをダウンロードしてコンパイルします。

OpenOCD をインストールするには

1. 次のコマンドを実行して libusb-1.0 をインストールします。

   ```
   sudo apt-get install libusb-1.0
   ```

2. http://openocd.org/ から openocd 0.9.0 ソースコードをダウンロードします。
3. openocd を抽出し、展開したディレクトリに移動します。
4. 次のコマンドで openocd を設定します。

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

5. make ユーティリティを実行して opencd をコンパイルします。

   ```
   make install
   ```

Eclipse の設定

> **Note**
>
> このセクションは、sudo の回避 のステップが完了していることを前提としています。

Eclipse は、アプリケーションの開発とデバッグに適した IDE です。スレッド対応のデバッグなど、統合されたデバッグサポートを備えた、ユーザーフレンドリーかつ充実した IDE を提供します。このセクションでは、サポートされているすべての開発ホストで共通の Eclipse 設定について説明します。

Eclipse を設定するには

1. Java ランタイム環境 (JRE) をダウンロードしてインストールします。

   Eclipse では JRE をインストールする必要があります。Eclipse のインストール後にインストールすることもできますが、最初にインストールすることをお勧めします。JRE のバージョン (32 ビットまたは 64 ビット) が Eclipse のバージョン (32 ビットまたは 64 ビット) と一致している必要があります。JRE は Oracle ウェブサイトの Java SE Runtime Environment 8 Downloads からダウンロードできます。
2. http://www.eclipse.org から「Eclipse IDE for C/C++ Developers」をダウンロードしてインストールします。Eclipse バージョン 4.9.0 以降がサポートされています。ダウンロードしたアーカイブを抽出するだけでインストールできます。プラットフォーム固有の Eclipse 実行可能ファイルを実行して、アプリケーションを起動します。

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモプロジェクトを実行するには、次の 2 つの方法があります。

- コマンドラインを使用します。
- Eclipse IDE を使用します。

このトピックでは、両方のオプションについて説明します。

プロビジョニング

- テストアプリケーションとデモアプリケーションのどちらを使用するかに応じて、次のいずれかのファイルにプロビジョニングデータを設定します。
  - `./tests/common/include/aws_clientcredential.h`
  - `./demos/common/include/aws_clientcredential.h`

  例:

```
#define clientcredentialWIFI_SSID "Wi-Fi SSID"
#define clientcredentialWIFI_PASSWORD "Wi-Fi password"
#define clientcredentialWIFI_SECURITY "Wi-Fi security"
```

  > **Note**
  >
  > 次の Wi-Fi セキュリティ値を入力できます。
  >
  > - `eWiFiSecurityOpen`
  > - `eWiFiSecurityWEP`
  > - `eWiFiSecurityWPA`
  > - `eWiFiSecurityWPA2`
  >
  > SSID とパスワードは、二重引用符で囲む必要があります。

コマンドラインを使用して FreeRTOS デモを構築して実行する

1. 次のコマンドを使用して、デモアプリケーションの構築を開始します。

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
DAFR_ENABLE_TESTS=0
```

   次の例と同じ出力が得られることを確認します。

```
marvell@pe-lt586:amzsdk$
marvell@pe-lt586:amzsdk$ cmake -DVENDOR=marvell -DBOARD=mw300_rd -DCOMPILER=arm-gcc -S . -
Bbuild -DAFR_ENABLE_TESTS=0

====================Configuration for Amazon FreeRTOS====================
  Version:                 v1.2.4
  Git version:             AMZSDK_V1.2.r6.p1-12-gdd17d10

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:                      Linux-4.15.0-47-generic
  Toolchain:               arm-gcc
  Toolchain path:          /home/marvell/Software/gcc-arm-none-eabi-4_9-2015q3
  CMake generator:         Unix Makefiles

Amazon FreeRTOS modules:
  Modules to build:        kernel, freertos_plus_tcp, bufferpool, crypto, greengrass, mqtt
, ota, pkcs11, secure_sockets, shadow, tls, wifi
  Disabled by user:
  Disabled by dependency:  posix

  Available demos:         demo_key_provisioning, demo_logging, demo_mqtt_hello_world, dem
o_mqtt_pubsub, demo_tcp, demo_shadow, demo_greengrass, demo_ota
  Available tests:         test_crypto, test_greengrass, test_mqtt, test_ota, test_pkcs11,
 test_secure_sockets, test_tls, test_shadow, test_wifi, test_memory
=========================================================================

-- Configuring done
-- Generating done
-- Build files have been written to: /home/marvell/gerrit/amzsdk/build
marvell@pe-lt586:amzsdk$
```

2. ビルドディレクトリに移動します。

```
cd build
```

3. make ユーティリティを実行して、アプリケーションを構築します。

```
make all -j4
```

次の図と同じ出力が得られることを確認します。



4. 次のコマンドを使用して、テストアプリケーションを構築します。

```
cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
DAFR_ENABLE_TESTS=1
cd build
make all -j4
```

aws_demos project と aws_tests project の切り替えを行うたびに cmake コマンドを実行します。

5. ファームウェアイメージを開発ボードのフラッシュに書き込みます。ファームウェアは、開発ボードがリセットされた後に実行されます。イメージをマイクロコントローラにフラッシュする前に SDK を構築する必要があります。

   a. ファームウェアイメージをフラッシュする前に、共通コンポーネントであるレイアウトと Boot2 を使用して開発ボードのフラッシュを準備します。次のコマンドを使用します。

   ```
   cd amzsdk_bundle-x.y.z
   ./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py -l ./vendors/
   marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt --boot2 ./vendors/
   marvell/WMSDK/mw320/boot2/bin/boot2.bin
   ```

   flashprog コマンドを実行すると、次の処理が開始されます。

- レイアウト — flashprog ユーティリティは、最初にレイアウトをフラッシュに書き込むように指示されます。レイアウトは、フラッシュのパーティション情報に似たものです。デフォルトのレイアウトは、/lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt にあります。
- Boot2 — WMSDK で使用されるブートローダーです。また、flashprog コマンドでも、ブートローダーがフラッシュに書き込まれます。ブートローダーは、ボードにフラッシュされた後、マイクロコントローラーのファームウェアイメージをロードします。下の図と同じ出力が得られることを確認します。

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245458s (115.728 KiB/s)
verified 29088 bytes in 0.350004s (81.160 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Erasing primary flash...done
Writing new flash layout...done
Writing "boot2" @0x0 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

b. ファームウェアはその機能に Wi-Fi チップセットを使用します。Wi-Fi チップセットには独自のファームウェアがあり、フラッシュにも存在する必要があります。flashprog.py ユーティリティを使用して、Boot2 ブートローダーと MCU ファームウェアをフラッシュしたのと同じ方法で Wi-Fi ファームウェアをフラッシュします。以下のコマンドを使用して、Wi-Fi ファームウェアをフラッシュします。

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

以下の図のようなコマンド出力になります。

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245498s (115.709 KiB/s)
verified 29088 bytes in 0.350229s (81.108 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "wififw" @0x12a000 (primary).....done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

c. 次のコマンドを使用して、MCU ファームウェアをフラッシュします。

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_demos.bin -r
```

d. ボードをリセットします。デモアプリケーションのログが表示されます。

e. テストアプリケーションを実行するには、同じディレクトリにある aws_tests.bin バイナリをフラッシュします。

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_tests.bin -r
```

コマンドの出力は、下の図と似たものになります。

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245499s (115.708 KiB/s)
verified 29088 bytes in 0.350231s (81.107 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "mcufw" @0x6a000 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
Resetting board...
Using OpenOCD interface file ftdi.cfg
Open On-Chip Debugger 0.9.0 (2015-07-15-15:28)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 3000 kHz
adapter_nsrst_delay: 100
Info : auto-selecting first available session transport "jtag". To override use 'transport
 select <transport>'.
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
sh_load
Info : clock speed 3000 kHz
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
Info : wmcore.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
shutdown command invoked
Resetting board done...
```

6. ファームウェアをフラッシュし、ボードをリセットすると、下の図のようにデモアプリケーションが開始されます。

```
Network connection successful.

Wi-Fi Connected to AP. Creating tasks which use network...
2 6293 [Startup Hook] Write certificate...
3 6296 [Startup Hook] Write device private key...
4 6362 [Startup Hook] Creating MQTT Echo Task...
6 11668 [MQTTEcho] MQTT echo connected.to connect to a2wtm15blvjjs8-ats.iot.us-east-2.amazonaws.com
7 11668 [MQTTEcho] MQTT echo test echoing task created.
8 11961 [MQTTEcho] MQTT Echo demo subscribed to freertos/demos/echo
9 12248 [MQTTEcho] Echo successfully published 'Hello World 0'
10 12591 [Echoing] Message returned with ACK: 'Hello World 0 ACK'
11 17633 [MQTTEcho] Echo successfully published 'Hello World 1'
12 17927 [Echoing] Message returned with ACK: 'Hello World 1 ACK'
13 22953 [MQTTEcho] Echo successfully published 'Hello World 2'
14 23276 [Echoing] Message returned with ACK: 'Hello World 2 ACK'
15 28245 [MQTTEcho] Echo successfully published 'Hello World 3'
16 28575 [Echoing] Message returned with ACK: 'Hello World 3 ACK'
17 33542 [MQTTEcho] Echo successfully published 'Hello World 4'
18 33980 [Echoing] Message returned with ACK: 'Hello World 4 ACK'
19 38823 [MQTTEcho] Echo successfully published 'Hello World 5'
20 39279 [Echoing] Message returned with ACK: 'Hello World 5 ACK'
21 44139 [MQTTEcho] Echo successfully published 'Hello World 6'
22 44501 [Echoing] Message returned with ACK: 'Hello World 6 ACK'
23 49516 [MQTTEcho] Echo successfully published 'Hello World 7'
24 50270 [Echoing] Message returned with ACK: 'Hello World 7 ACK'
25 54796 [MQTTEcho] Echo successfully published 'Hello World 8'
26 55129 [Echoing] Message returned with ACK: 'Hello World 8 ACK'
27 60080 [MQTTEcho] Echo successfully published 'Hello World 9'
28 60389 [Echoing] Message returned with ACK: 'Hello World 9 ACK'
29 65378 [MQTTEcho] Echo successfully published 'Hello World 10'
30 65998 [Echoing] Message returned with ACK: 'Hello World 10 ACK'
31 70658 [MQTTEcho] Echo successfully published 'Hello World 11'
32 70964 [Echoing] Message returned with ACK: 'Hello World 11 ACK'
33 75958 [MQTTEcho] MQTT echo demo finished.
34 75958 [MQTTEcho] ----Demo finished----
```

7. (オプション) イメージをテストする別の方法として、flashprog ユーティリティを使用して、ホストからマイクロコントローラー RAM に直接マイクロコントローラーイメージをコピーします。イメージはフラッシュにコピーされないため、マイクロコントローラを再起動すると失われます。

   ファームウェアイメージを SRAM にロードすると実行ファイルがすぐに起動されるため、処理が高速になります。このメソッドは、主に反復的な開発で使用されます。

   次のコマンドを使用して、ファームウェアを SRAM にロードします。

   ```
   cd amzsdk_bundle-x.y.z
   ./lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/OpenOCD/ramload.py
    build/cmake/vendors/marvell/mw300_rd/aws_demos.axf
   ```

   次のようなコマンド出力が表示されます。

```
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
```

コマンドの実行が完了すると、デモアプリケーションのログが表示されます。

Eclipse IDE を使用して FreeRTOS デモを構築して実行する

1. Eclipse ワークスペースをセットアップする前に、cmake コマンドを実行する必要があります。

   次のコマンドを実行して、aws_demos Eclipse プロジェクトを操作します。

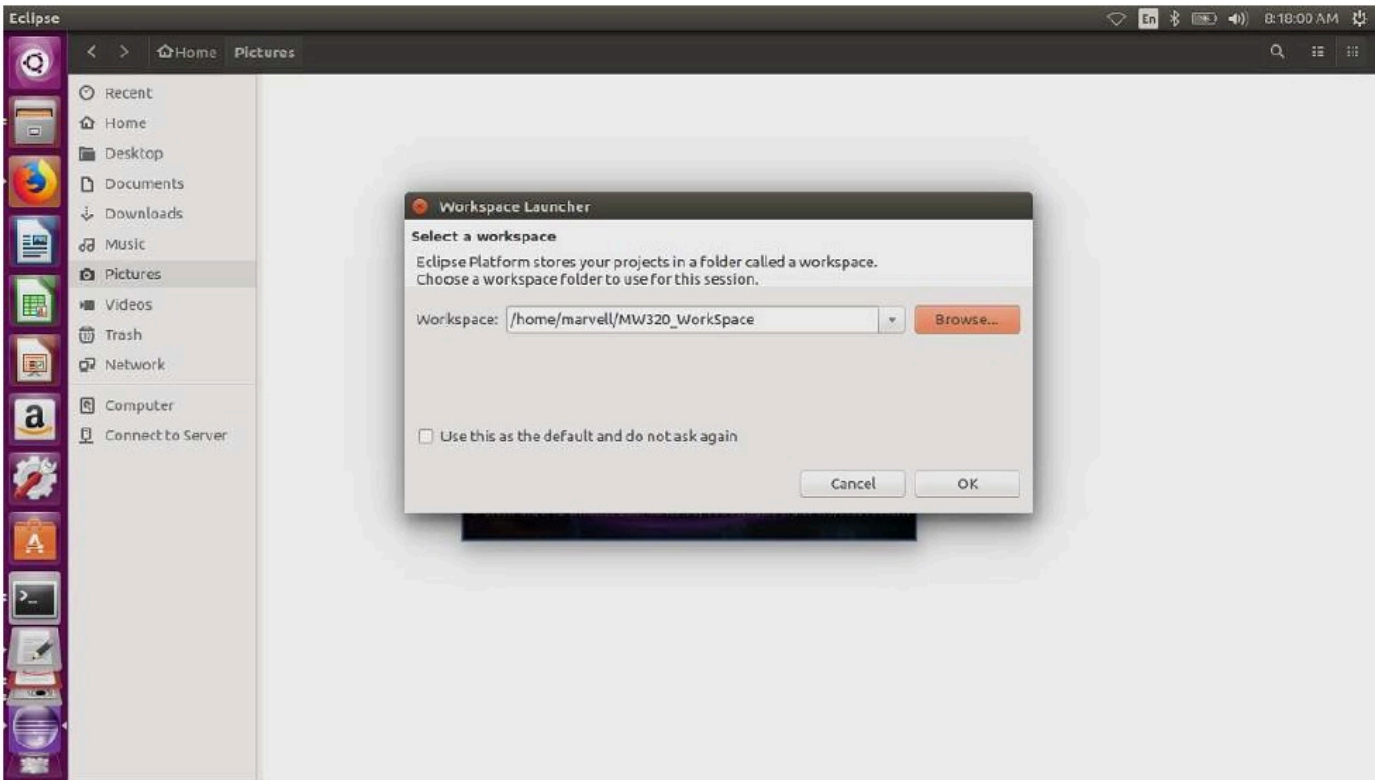
   ```
   cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
   DAFR_ENABLE_TESTS=0
   ```

   次のコマンドを実行して、aws_tests Eclipse プロジェクトを操作します。

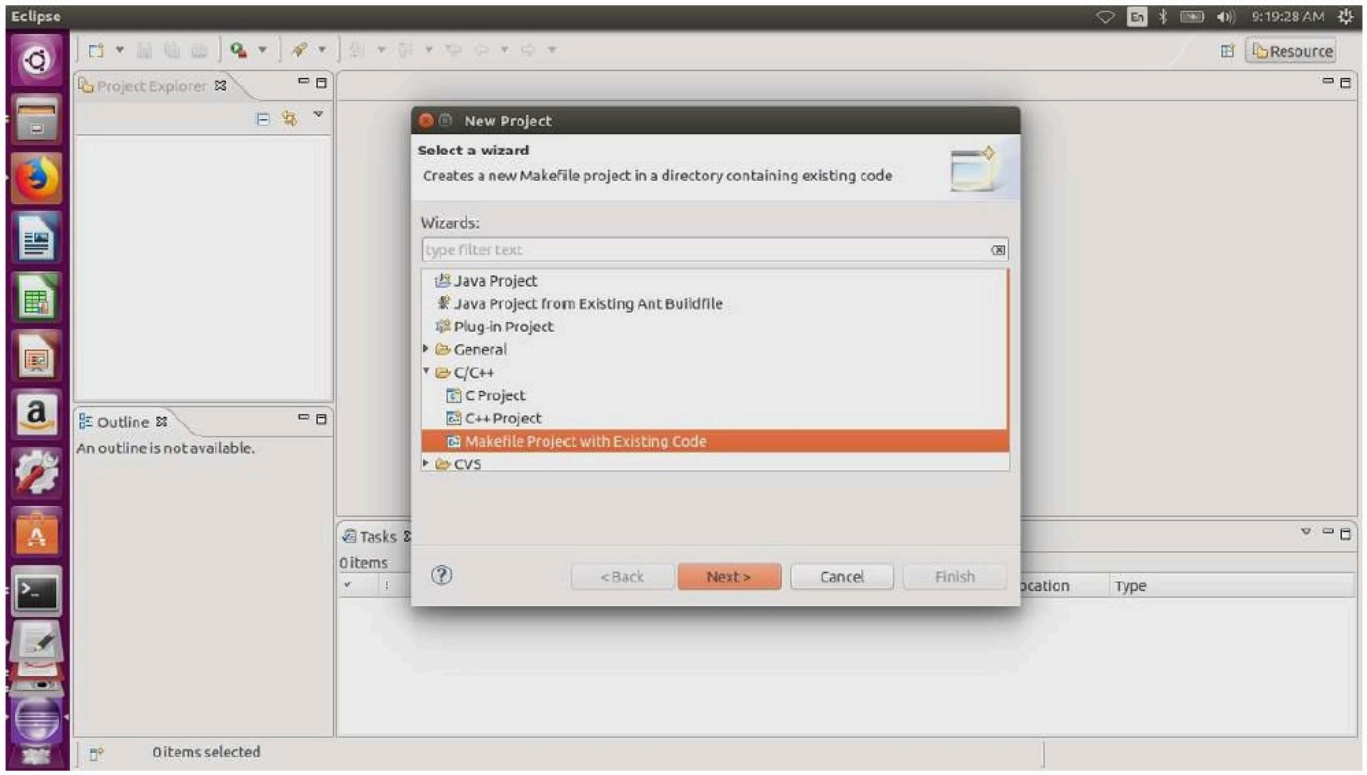
   ```
   cmake -DVENDOR=marvell -DBOARD=mw320 -DCOMPILER=arm-gcc -S .  -Bbuild -
   DAFR_ENABLE_TESTS=1
   ```

> **Tip**
>
> aws_demos プロジェクトと aws_tests プロジェクトの切り替えを行うたびに cmake コマンドを実行します。

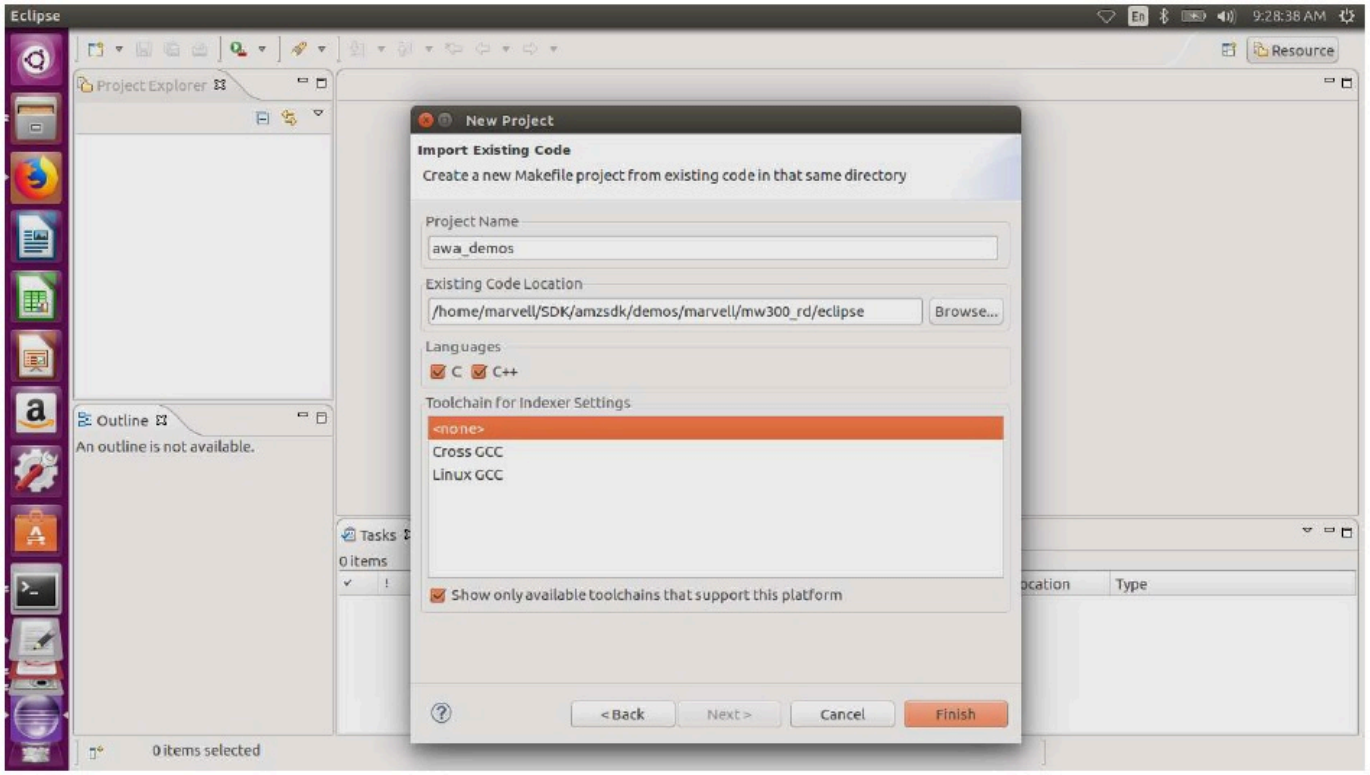
2. Eclipse を開き、プロンプトが表示されたら、次の図のように Eclipse ワークスペースを選択します。



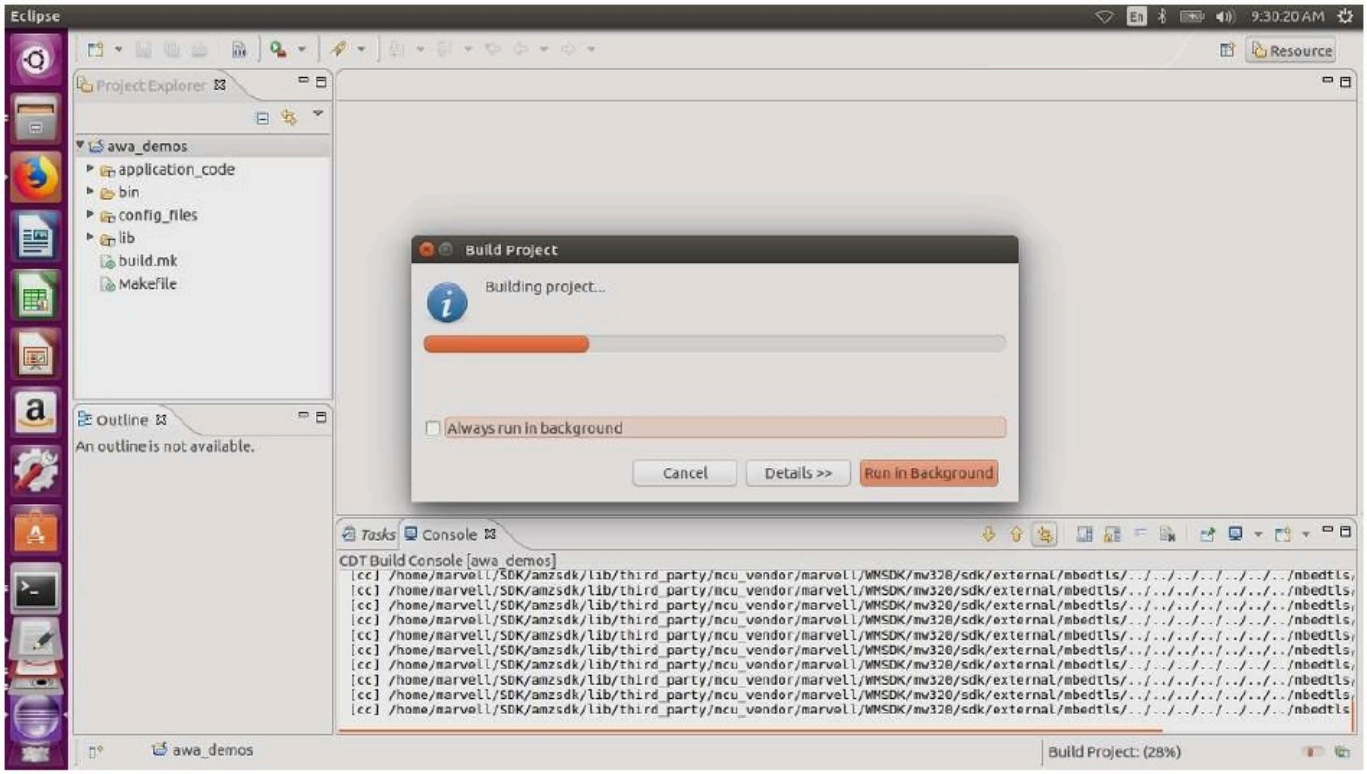
3. 下の図のようにオプションを選択して、[Makefile Project: with Existing Code] (Makefile プロジェクト: 既存のコードを含む) を作成します。



4. [Browse] (参照) を選択し、既存のコードのディレクトリを指定し、[Finish] (完了) を選択します。



5. ナビゲーションペインで、プロジェクトエクスプローラーの aws_demos を選択します。aws_demos を右クリックしてメニューを開き、[Build] (構築) を選択します。



構築が成功すると、build/cmake/vendors/marvell/mw300_rd/aws_demos.bin ファイルが生成されます。

6. コマンドラインツールを使用して、レイアウトファイル (layout.txt)、Boot2 バイナリ (boot2.bin)、MCU ファームウェアバイナリ (aws_demos.bin)、Wi-Fi ファームウェアをフラッシュします。

   a. ファームウェアイメージをフラッシュする前に、共通のコンポーネントであるレイアウトと Boot2 を使用した開発ボードのフラッシュを準備します。次のコマンドを使用します。

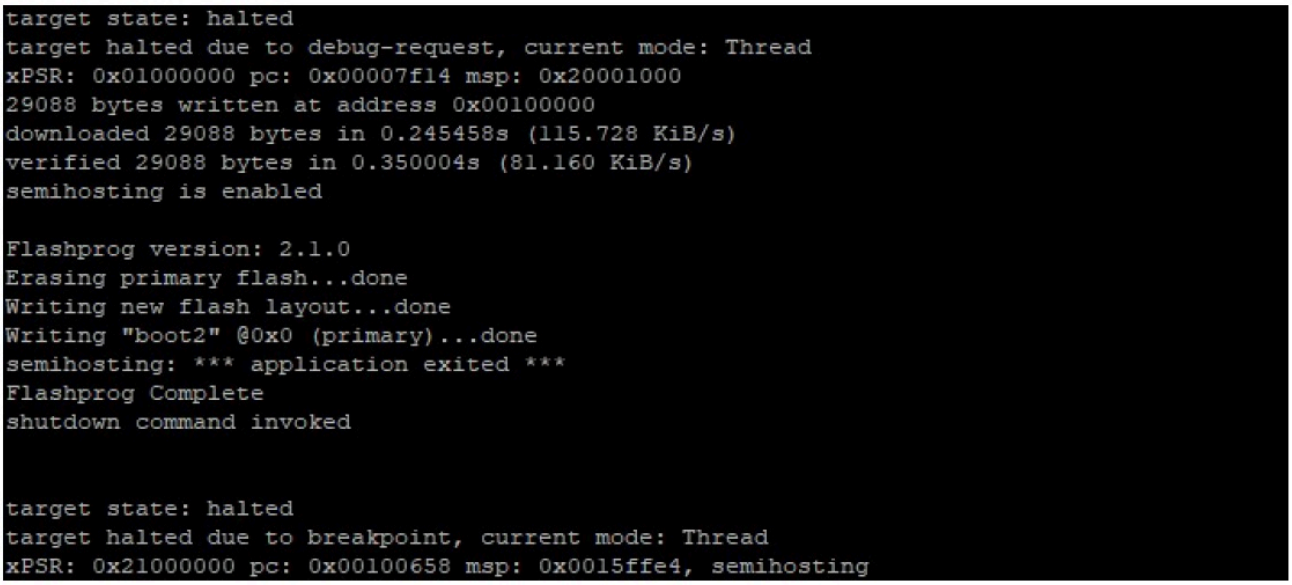
```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py -l ./vendors/
marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt --boot2 ./vendors/
marvell/WMSDK/mw320/boot2/bin/boot2.bin
```

   flashprog コマンドを実行すると、次の処理が開始されます。

   - レイアウト — flashprog ユーティリティは、最初にレイアウトをフラッシュに書き込むように指示されます。レイアウトは、フラッシュのパーティション情報に似たものです。

デフォルトのレイアウトは、/lib/third_party/mcu_vendor/marvell/WMSDK/mw320/sdk/tools/OpenOCD/mw300/layout.txt にあります。

- Boot2 — WMSDK で使用されるブートローダーです。また、flashprog コマンドでも、ブートローダーがフラッシュに書き込まれます。ブートローダーは、フラッシュされた後、マイクロコントローラーのファームウェアイメージをロードします。下の図と同じ出力が得られることを確認します。



b. ファームウェアはその機能に Wi-Fi チップセットを使用します。Wi-Fi チップセットには独自のファームウェアがあり、フラッシュにも存在する必要があります。flashprog.py ユーティリティを使用して、Boot2 ブートローダーと MCU ファームウェアをフラッシュしたのと同じ方法で Wi-Fi ファームウェアをフラッシュします。以下のコマンドを使用して、Wi-Fi ファームウェアをフラッシュします。

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

以下の図のようなコマンド出力になります。

```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245498s (115.709 KiB/s)
verified 29088 bytes in 0.350229s (81.108 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "wififw" @0x12a000 (primary).....done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
```

c. 次のコマンドを使用して、MCU ファームウェアをフラッシュします。

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_demos.bin -r
```

d. ボードをリセットします。デモアプリケーションのログが表示されます。

e. テストアプリケーションを実行するには、同じディレクトリにある aws_tests.bin バイナリをフラッシュします。

```
cd amzsdk_bundle-x.y.z
./vendors/marvell/WMSDK/mw320/sdk/tools/OpenOCD/flashprog.py --mcufw build/
cmake/vendors/marvell/mw300_rd/aws_tests.bin -r
```

コマンドの出力は、下の図と似たものになります。

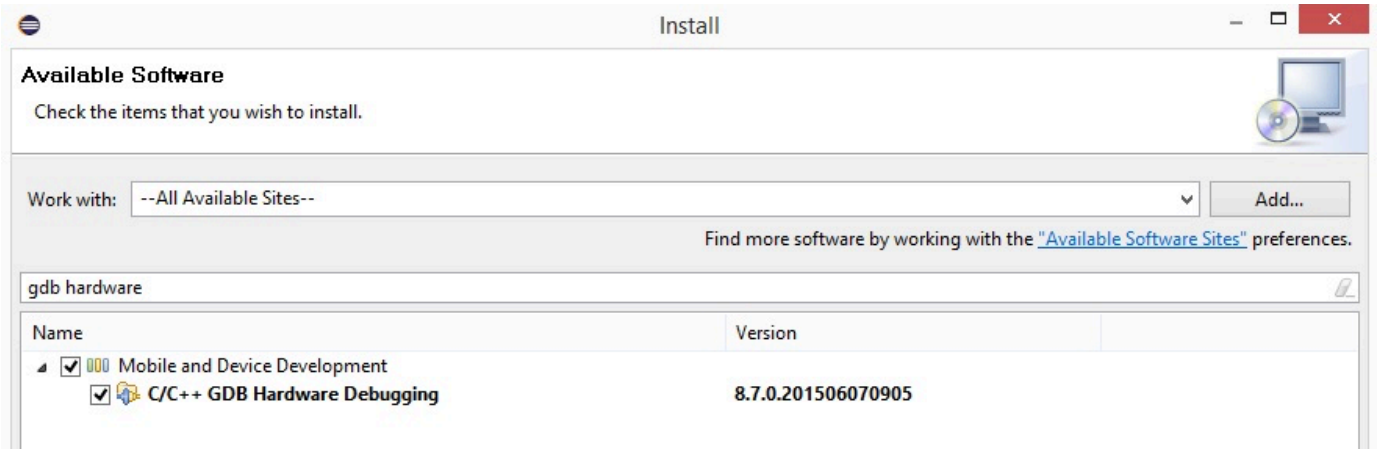
```
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00007f14 msp: 0x20001000
29088 bytes written at address 0x00100000
downloaded 29088 bytes in 0.245499s (115.708 KiB/s)
verified 29088 bytes in 0.350231s (81.107 KiB/s)
semihosting is enabled

Flashprog version: 2.1.0
Writing "mcufw" @0x6a000 (primary)...done
semihosting: *** application exited ***
Flashprog Complete
shutdown command invoked


target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x21000000 pc: 0x00100658 msp: 0x0015ffe4, semihosting
Resetting board...
Using OpenOCD interface file ftdi.cfg
Open On-Chip Debugger 0.9.0 (2015-07-15-15:28)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 3000 kHz
adapter_nsrst_delay: 100
Info : auto-selecting first available session transport "jtag". To override use 'transport
 select <transport>'.
jtag_ntrst_delay: 100
cortex_m reset_config sysresetreq
sh_load
Info : clock speed 3000 kHz
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
Info : wmcore.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: wmcore.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0
x4)
shutdown command invoked
Resetting board done...
```

デバッグ

- Eclipse を起動し、[Help] (ヘルプ) を選択してから、[Install New Software] (新しいソフトウェアのインストール) を選択します。[Work with] (操作) メニューで、[All Available Sites] (利用可能なすべてのサイト) を選択します。フィルターテキスト `GDB Hardware` を入力します。[C/C++ GDB Hardware Debugging] (C/C++ GDB ハードウェアデバッグ) オプションを選択して、プラグインをインストールします。



トラブルシューティング

ネットワークの問題

ネットワーク認証情報を確認します。FreeRTOS デモプロジェクトを構築して実行する の「プロビジョニング」を参照してください。

追加ログの有効化

- ボード固有のログを有効にします。

  main.c ファイルの prvMiscInitialization 関数の wmstdio_init(UART0_ID, 0) への呼び出しを、テストまたはデモ用に有効にします。

- Wi-Fi ログの有効化

  freertos/vendors/marvell/WMSDK/mw320/sdk/src/incl/autoconf.h ファイルのマクロ CONFIG_WLCMGR_DEBUG を有効にします。

GDB の使用

SDK と一緒にパッケージ化された arm-none-eabi-gdb および gdb コマンドファイルを使用することをお勧めします。ディレクトリに移動します。

```
cd freertos/lib/third_party/mcu_vendor/marvell/WMSDK/mw320
```

GDB に接続するには、次のコマンドを (1 行で) 実行します。

```
arm-none-eabi-gdb -x ./sdk/tools/OpenOCD/gdbinit ../../../../../../build/cmake/
vendors/marvell/mw300 _rd/aws_demos.axf
```

MediaTek MT7697Hx Development Kit の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、MediaTek MT7697Hx 開発キットの使用を開始するための手順について説明します。MediaTek MT7697Hx Development Kit がない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
4. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

開発環境をセットアップする

環境をセットアップする前に、MediaTek MT7697Hx Development Kit の USB ポートにコンピュータを接続します。

Keil MDK をダウンロードしてインストールする

ボードで FreeRTOS プロジェクトを設定、構築、実行するには、GUI ベースの Keil Microcontroller Development Kit (MDK) を使用します。Keil MDK には、µVision IDE および µVision Debugger が含まれています。

> **Note**
>
> Keil MDK は、Windows 7、Windows 8、および Windows 10 64 ビットコンピュータでのみサポートされています。

Keil MDK をダウンロードしてインストールするには

1. Keil MDK の開始方法ページを開き、[Download MDK-Core] (MDK-Core のダウンロード) を選択します。
2. Keil に登録するには、情報を入力して送信します。
3. MDK 実行ファイルを右クリックし、Keil MDK インストーラをコンピュータに保存します。
4. Keil MDK インストーラを開き、ステップを完了します。MediaTek デバイスパック (MT76x7 シリーズ) がインストールされていることを確認します。

シリアル接続の確立

USB ケーブルを使用して、ボードをホストコンピュータに接続します。Windows デバイスマネージャーに COM ポートが表示されます。デバッグするには、ターミナルユーティリティツール (例: HyperTerminal や TeraTerm) を使用して、ポートへのセッションを開きます。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*****example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

Keil MDK を使用して FreeRTOS デモプロジェクトを構築して実行する

Keil µVision で FreeRTOS デモプロジェクトを構築するには

1. [Start] (スタート) メニューから Keil µVision 5 を開きます。
2. `projects/mediatek/mt7697hx-dev-kit/uvision/aws_demos/aws_demos.uvprojx` プロジェクトファイルを開きます。
3. メニューから、[Project] (プロジェクト)、[Build target] (ビルドターゲット) の順に選択します。

   コードがビルドされたら、`projects/mediatek/mt7697hx-dev-kit/uvision/aws_demos/out/Objects/aws_demo.axf` にデモ実行ファイルが表示されます。

FreeRTOS デモプロジェクトを実行するには

1. MediaTek MT7697Hx Development Kit を PROGRAM モードに設定します。

   kit を PROGRAM モードに設定するには、[PROG] ボタンを押し続けます。[PROG] ボタンを押し続けながら、[RESET] (リセット) ボタンを押して離し、[PROG] ボタンを離します。
2. メニューから、[Flash]、[Configure Flash Tools] (Flash ツールを設定) を選択します。
3. [Options for Target '**aws_demo**'] で、[デバッグ] タブを選択します。[Use] (使用) を選択して、デバッガーを [CMSIS-DAP Debugger] (CMSIS-DAP デバッガー) に設定し、[OK] を選択します。
4. メニューから、[Flash]、[Download] (ダウンロード) を選択します。

   ダウンロードが完了すると、µVision より通知されます。
5. ターミナルユーティリティを使用して、シリアルコンソールウィンドウを開きます。シリアルポートを 115200 bps、パリティなし、8 ビット、および 1 ストップビットに設定します。
6. MediaTek MT7697Hx Development Kit で [RESET] (リセット) ボタンを選択します。

トラブルシューティング

Keil µVision での FreeRTOS プロジェクトのデバッグ

現時点では、Keil µVision で MediaTek 用の FreeRTOS デモプロジェクトをデバッグする前に、Keil µVision に付属している MediaTek パッケージを編集する必要があります。

MediaTek パッケージを編集して FreeRTOS プロジェクトをデバッグするには

1. Keil MDK インストールフォルダの `Keil_v5\ARM\PACK\.Web\MediaTek.MTx.pdsc` ファイルを開きます。
2. `flag = Read32(0x20000000);` のインスタンスをすべて、`flag = Read32(0x0010FBFC);` に置き換えます。
3. `Write32(0x20000000, 0x76877697);` のインスタンスをすべて、`Write32(0x0010FBFC, 0x76877697);` に置き換えます。

プロジェクトのデバッグを開始するには

1. メニューから、[Flash]、[Configure Flash Tools] (Flash ツールを設定) を選択します。
2. [Target] (ターゲット) タブを選択し、[Read/Write Memory Areas] (読み取り/書き取りメモリエリア) を選択します。IRAM1 と IRAM2 がいずれも選択されていることを確認します。
3. [Debug] (デバッグ) タブ、[CMSIS-DAP Debugger] (CMSIS-DAP デバッガー) の順に選択します。
4. `vendors/mediatek/boards/mt7697hx-dev-kit/aws_demos/application_code/main.c` を開き、マクロ `MTK_DEBUGGER` を `1` に設定します。
5. µVision でデモプロジェクトを再構築します。
6. MediaTek MT7697Hx Development Kit を PROGRAM モードに設定します。

   kit を PROGRAM モードに設定するには、[PROG] ボタンを押し続けます。[PROG] ボタンを押し続けながら、[RESET] (リセット) ボタンを押して離し、[PROG] ボタンを離します。
7. メニューから、[Flash]、[Download] (ダウンロード) を選択します。

   ダウンロードが完了すると、µVision より通知されます。
8. MediaTek MT7697Hx Development Kit で [RESET] (リセット) ボタンを選択します。
9. µVision メニューから、[Debug] (デバッグ)、[Start/Stop Debug Session] (デバッグセッションの開始/停止) の順に選択します。デバッグセッションを開始すると、[Call Stack + Locals] (スタックの呼び出し + ローカル) ウィンドウが開きます。
10. メニューから [Debug] (デバッグ) を選択し、[Stop] (停止) を選択して、コードの実行を一時停止します。プログラムカウンタは次の行で停止します。

```
{ volatile int wait_ice = 1 ; while ( wait_ice ) ; }
```

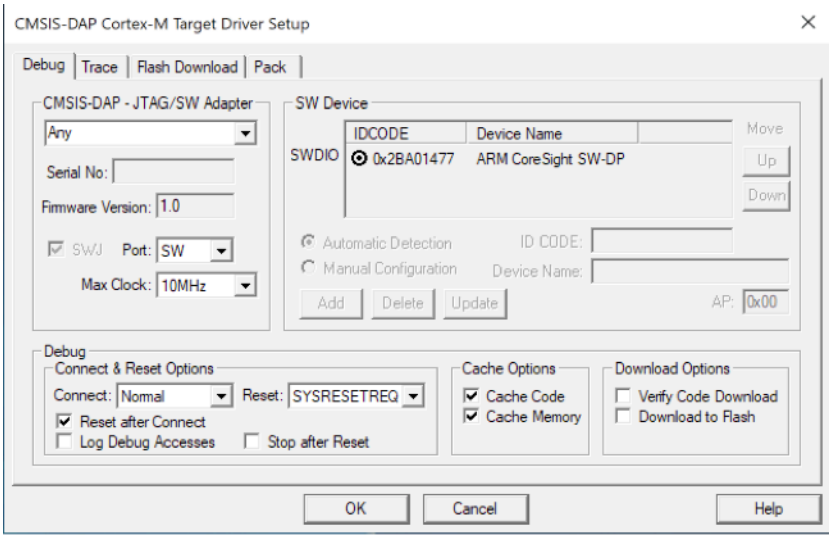
11. [Call Stack + Locals] (スタックの呼び出し + ローカル) ウィンドウで、`wait_ice` の値を `0` に変更します。
12. プロジェクトのソースコードのブレークポイントを設定し、コードを実行します。

IDE デバッガー設定のトラブルシューティング

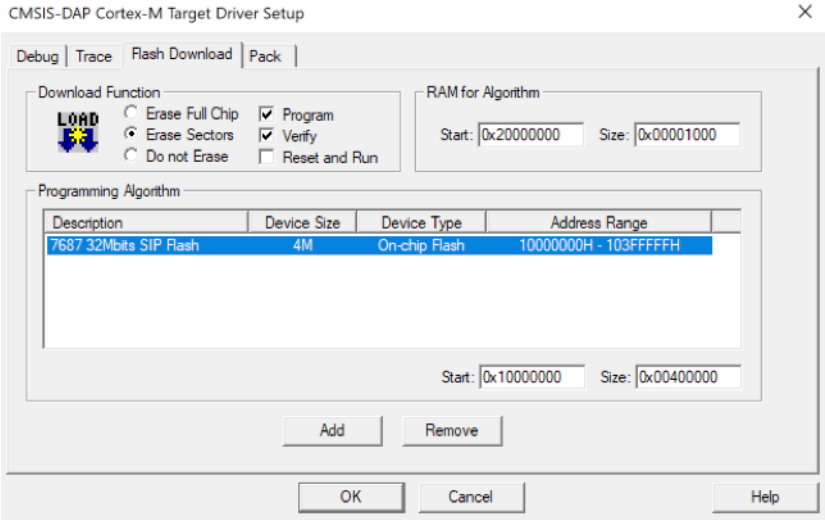
アプリケーションのデバッグ中に問題が発生した場合は、デバッガー設定が正しくない可能性があります。

デバッガー設定が正しいことを確認するには

1. Keil µVision を開きます。
2. `aws_demos` プロジェクトを右クリックし [Options] (オプション) を選択し、[Utilities] (ユーティリティ) タブで [-- Use Debug Driver --] (デバッグドライバを使用) の隣にある [Settings] (設定) を選択します。
3. [Debug] (デバッグ) タブの設定が以下のように表示されていることを確認します。



4. [Flash Download] (フラッシュダウンロード) タブの設定が以下のように表示されていることを確認します。



FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Microchip Curiosity PIC32MZ EF の開始方法

Important

このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

Note

Microchip との合意に基づき、Curiosity PIC32MZEF (DM320104) は FreeRTOS Reference Integration リポジトリのメインブランチから削除されるため、新しいリリースには搭載されません。Microchip は PIC32MZEF (DM320104) が新しい設計に推奨されなくなったことを公式に発表しました。PIC32MZEF プロジェクトとソースコードには、以前のリリースタグから引き続きアクセスできます。Microchip では、新しい設計に Curiosity PIC32MZ-EF-2.0 Development board (DM320209) を使用するよう推奨しています。Pic32mzv1 プラットフォームは、FreeRTOS リファレンス統合リポジトリの v202012.00 に引き続き用意されています。ただし、プラットフォームは FreeRTOS リファレンスの v202107.00 によってサポートされなくなりました。

このチュートリアルでは、Microchip Curiosity PIC32MZ EF の使用を開始するための手順について説明します。Microchip Curiosity PIC32MZ EF バンドルがない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

このバンドルには、次の項目が含まれます。

- Curiosity PIC32MZ EF Development Board
- MikroElectronika USB UART click Board
- MikroElectronika WiFi 7 click Board
- PIC32 LAN8720 PHY daughter board

デバッグのため、以下も必要です。

- MPLAB Snap In-Circuit Debugger
- (オプション) PICkit 3 プログラミングケーブルキット

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。

> **Important**
>
> - このトピックでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。
> - *`freertos`* パスにスペース文字が含まれていると、構築が失敗する可能性があります。リポジトリをクローンまたはコピーするときは、作成するパスにスペース文字が含まれていないことを確認してください。
> - Microsoft Windows でのファイルパスの最大長は 260 文字です。FreeRTOS のダウンロードディレクトリパスが長くなると、構築が失敗する可能性があります。
> - ソースコードにはシンボリックリンクが含まれている可能性があるため、Windows を使用してアーカイブを抽出する場合は、次の操作を行う必要があります。
>   - 開発者モードを有効にするか、または、
>   - 管理者としてコンソールを使用します。
>
>   この操作を行えば、Windows でアーカイブを抽出する際にシンボリックリンクを適切に作成できます。この操作を行わないと、シンボリックリンクは、そのパスがテキストとして

含まれる、または空白の通常ファイルとして書き込まれます。詳細については、ブログの投稿「Symlinks in Windows 10!」を参照してください。

Windows で Git を使用する場合は、開発者モードを有効にするか、以下を実行する必要があります。

- 次のコマンドを使用して、`core.symlinks` を true に設定します。

```
git config --global core.symlinks true
```

- システムへの書き込みを行う git コマンド (git pull、git clone、git submodule update --init --recursive など) を使用する場合は、管理者としてコンソールを使用します。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. ボードをホストマシンに接続します。
2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
5. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

Microchip Curiosity PIC32MZ EF ハードウェアの設定

1. MikroElectronika USB UART click Board を Microchip Curiosity PIC32MZ EF の microBUS 1 コネクタに接続します。
2. PIC32 LAN8720 PHY daughter board を Microchip Curiosity PIC32MZ EF の J18 ヘッダーに接続します。
3. USB A to USB mini-B ケーブルを使用して、MikroElectronika USB UART click Board をコンピュータに接続します。
4. ボードをインターネットに接続するには、次のいずれかのオプションを使用します。

- Wi-Fi を使用するには、MikroElectronika Wi-Fi 7 click Board を Microchip Curiosity PIC32MZ EF の microBUS 2 コネクタに接続します。「FreeRTOS デモを設定する」を参照してください。
- イーサネットを使用するには、Microchip Curiosity PIC32MZ EF Board をインターネットに接続し、PIC32 LAN8720 PHY daughter board を Microchip Curiosity PIC32MZ EF の J18 ヘッダーに接続します。イーサネットケーブルの一端を LAN8720 PHY ドーターボードに接続します。他方をルーターまたはその他のインターネットに接続します。プリプロセッサマクロ `PIC32_USE_ETHERNET` も定義する必要があります。

5. まだ行っていない場合は、Microchip Curiosity PIC32MZ EF の ICSP ヘッダーにアングルコネクタをはんだ付けします。
6. ICSP ケーブルの一端を PICkit 3 Programming Cable Kit から Microchip Curiosity PIC32MZ EF に接続します。

   PICkit 3 Programming Cable Kit がない場合は、M-F Dupont ワイヤージャンパーを使用して代わりに接続を配線できます。白い円は、ピン 1 の位置を示していることに注意してください。
7. ICSP ケーブル (またはジャンパー) のもう一方の端を MPLAB Snap Debugger に接続します。8 ピン SIL プログラミングコネクタのピン 1 は、ボードの右下の黒い三角形でマークされています。

   Microchip Curiosity PIC32MZ EF のピン 1 へのケーブル (白い円で示す) が、MPLAB Snap Debugger のピン 1 に合っていることを確認します。

   MPLAB Snap Debugger の詳細については、MPLAB Snap In-Circuit Debugger 情報シートを参照してください。

PICkit On Board (PKOB) を使用した Microchip Curiosity PIC32MZ EF ハードウェアのセットアップ

前のセクションのセットアップ手順に従うことをお勧めします。ただし、次の手順に従って、統合された PICkit On Board (PKOB) プログラマー/デバッガーを使用して、基本的なデバッグで FreeRTOS デモを評価および実行できます。

1. MikroElectronika USB UART click Board を Microchip Curiosity PIC32MZ EF の microBUS 1 コネクタに接続します。
2. ボードをインターネットに接続するには、次のいずれかの操作を実行します。

- Wi-Fi を使用するには、MikroElectronika Wi-Fi 7 click Board を Microchip Curiosity PIC32MZ EF の microBUS 2 コネクタに接続します。(「FreeRTOS デモを設定する」の「Wi-Fi を設定するには」の手順に従います)。
- イーサネットを使用するには、Microchip Curiosity PIC32MZ EF Board をインターネットに接続し、PIC32 LAN8720 PHY daughter board を Microchip Curiosity PIC32MZ EF の J18 ヘッダーに接続します。イーサネットケーブルの一端を LAN8720 PHY ドーターボードに接続します。他方をルーターまたはその他のインターネットに接続します。プリプロセッサマクロ `PIC32_USE_ETHERNET` も定義する必要があります。

3. USB type A to USB micro-B ケーブルを使用して、Microchip Curiosity PIC32MZ EF Board の「USB DEBUG」という名前の USB micro-B ポートをコンピュータに接続します。
4. USB A to USB mini-B ケーブルを使用して、MikroElectronika USB UART click Board をコンピュータに接続します。

開発環境をセットアップする

> **Note**
>
> このデバイスの FreeRTOS プロジェクトは、MPLAB Harmony v2 に基づいています。プロジェクトをビルドするには、Harmony v2 と互換性のある MPLAB ツールのバージョン (MPLAB XC32 コンパイラの v2.10、MPLAB Harmony Configurator (MHC) のバージョン 2.X.X など) を使用する必要があります。

1. Python バージョン 3.x 以降をインストールします。
2. MPLAB X IDE をインストールします。

   > **Note**
   >
   > FreeRTOS AWS リファレンス統合 v202007.00 は現在、MPLabv5.35 でのみサポートされています。FreeRTOS AWS リファレンス統合の以前のバージョンは、MPLabv5.40.

   MPLabv5.35 ダウンロード

   - MPLAB X Integrated Development Environment for Windows
   - MPLAB X Integrated Development Environment for macOS

- MPLAB X Integrated Development Environment for Linux

最新の MPLab ダウンロード (MPLabv5.40)

- MPLAB X Integrated Development Environment for Windows
- MPLAB X Integrated Development Environment for macOS
- MPLAB X Integrated Development Environment for Linux

3. MPLAB XC32 コンパイラをインストールします。

- MPLAB XC32/32++ Compiler for Windows
- MPLAB XC32/32++ Compiler for macOS
- MPLAB XC32/32++ Compiler for Linux

4. UART ターミナルエミュレーターを起動し、以下の設定で接続を開きます。

- ボーレート: 115200
- データ: 8 ビット
- パリティ: なし
- ストップビット: 1
- フロー制御: なし

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS デモプロジェクトを構築して実行する

MPLAB IDE で FreeRTOS デモを開く

1. MPLAB IDE を開きます。複数のバージョンのコンパイラをインストールしている場合は、IDE 内から使用するコンパイラを選択する必要があります。
2. [File] (ファイル) メニューで、[Open Project] (プロジェクトを開く) を選択します。
3. `projects/microchip/curiosity_pic32mzef/mplab/aws_demos` に移動して開きます。
4. [Open project] (プロジェクトを開く) を選択します。

> **Note**
>
> プロジェクトを初めて開く場合は、コンパイラに関するエラーメッセージが表示される可能性があります。IDE で、[Tools] (ツール)、[Options] (オプション)、[Embedded] (埋め込み) に移動して、プロジェクトに使用しているコンパイラを選択します。

イーサネットを使用して接続するには、プリプロセッサマクロ `PIC32_USE_ETHERNET` を定義する必要があります。

MPLAB IDE を使用してイーサネットで接続するには

1. MPLAB IDE でプロジェクトを右クリックし、[Properties] (プロパティ) を選択します。
2. [Project Properties] (プロジェクトプロパティ) ダイアログボックスで、***compiler-name*** (グローバルオプション) を選択して展開し、***compiler-name***-gcc を選択します。
3. [Options categories] (オプションのカテゴリ) で、[Preprocessing and messages] (前処理とメッセージ) を選択し、[Preprocessor macros] (プリプロセッサマクロ) に `PIC32_USE_ETHERNET` 文字列を追加します。

FreeRTOS デモプロジェクトを実行する

1. プロジェクトを再構築します。
2. [Projects] (プロジェクト) タブで、`aws_demos` の最上位フォルダを右クリックし、[Debug] (デバッグ) を選択します。
3. デバッガーが `main()` のブレークポイントで停止したら、[Run] (実行) メニューから [Resume] (再開) を選択します。

CMake で FreeRTOS デモを構築する

FreeRTOS 開発に IDE を使用しない場合は、代わりに CMake を使用して、サードパーティのコードエディタおよびデバッグツールを使用して開発したデモアプリケーションまたはアプリケーションを構築して実行できます。

CMake で FreeRTOS デモを構築するには

1. 次のような生成されたビルドファイルを格納するディレクトリ (*build-directory* など) を作成します。
2. ソースコードからビルドファイルを生成するには、次のコマンドを使用します。

```
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**
>
> Hexmate およびツールチェーンバイナリへの適切なパス (`C:\Program Files (x86)\Microchip\MPLABX\v5.35\mplab_platform\bin`、`C:\Program Files\Microchip\xc32\v2.40\bin` パスなど) を指定する必要があります。

3. ディレクトリをビルドディレクトリ (*build-directory*) に変更して、そのディレクトリから `make` を実行します。

詳細については、「FreeRTOS で CMake を使用する」を参照してください。

イーサネットを使用して接続するには、プリプロセッサマクロ `PIC32_USE_ETHERNET` を定義する必要があります。

トラブルシューティング

トラブルシューティング情報については、「トラブルシューティングの開始方法」を参照してください。

Nordic nRF52840-DK の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Nordic nRF52840-DK の使用を開始するための手順について説明します。Nordic nRF52840-DK がない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

開始する前に、FreeRTOS Bluetooth Low Energy 用の AWS IoT と Amazon Cognito のセットアップすることが必要です。

FreeRTOS Bluetooth Low Energy デモを実行するには、Bluetooth および Wi-Fi 機能が搭載された iOS または Android デバイスも必要です。

> **Note**
>
> iOS デバイスを使用している場合は、デモ用モバイルアプリケーションを構築するために Xcode が必要です。Android デバイスを使用している場合、Android Studio を使用してデモ用モバイルアプリケーションを構築できます。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. ボードをホストマシンに接続します。
2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。
5. モニタリングおよびデバッグの目的で、シリアル接続経由でボード上で実行されているアプリケーションに接続します。

Nordic ハードウェアをセットアップする

ホストコンピュータを、Nordic nRF52840 ボードのコイン型電池ホルダーのすぐ上にある J2 という USB ポートに接続します。

Nordic nRF52840-DK のセットアップ方法の詳細については、nRF52840 Development Kit ユーザーガイドを参照してください。

開発環境をセットアップする

Segger Embedded Studio のダウンロードとインストール

FreeRTOS は、Nordic nRF52840-DK 用の開発環境として Segger Embedded Studio をサポートしています。

環境をセットアップするには、Segger Embedded Studio をホストコンピュータにダウンロードしてインストールする必要があります。

Segger Embedded Studio をダウンロードしてインストールするには

1. Segger Embedded Studio Downloads ページに移動し、お使いのオペレーティングシステムの ARM オプション用 Embedded Studio を選択します。
2. インストーラを実行し、プロンプトに従ってインストールを完了します。

FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションをセットアップする

Bluetooth Low Energy で FreeRTOS デモプロジェクトを実行するには、モバイルデバイスで FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションを実行する必要があります。

FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションをセットアップするには

1. モバイルプラットフォーム用の SDK をホストコンピュータにダウンロードしてインストールするには、「FreeRTOS Bluetooth デバイス用の Mobile SDK」の手順に従います。
2. モバイルデバイスにデモモバイルアプリケーションをセットアップするには、「FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション」の手順に慕います。

シリアル接続の確立

Segger Embedded Studio には、ターミナルエミュレーターが搭載されており、ボードにシリアル接続してログメッセージを受信することができます。

Segger Embedded Studio とのシリアル接続を確立するには

1. Segger Embedded Studio を開きます。
2. トップメニューから、[Target] (ターゲット)、[Connect J-Link] (J-Link を接続) の順に選択します。
3. トップメニューで [Tools] (ツール)、[Terminal Emulator] (ターミナルエミュレータ)、[Properties] (プロパティ) を選択し、ターミナルエミュレーターをインストールする の指示に従ってプロパティを設定します。
4. トップメニューで、[Tools] (ツール)、[Terminal Emulator] (ターミナルエミュレータ)、[Connect *port* (115200,N,8,1)] (ポート (115200,N,8,1) に接続) の順に選択します。

> **Note**
>
> Segger 組み込みスタジオターミナルエミュレータは、入力機能をサポートしていません。このため、PuTTy、Tera Term、GNU Screen などのターミナルエミュレータを使用してください。ターミナルエミュレーターをインストールする の手順に従って、ターミナルをシリアル接続でボードに接続するよう設定します。

FreeRTOS をダウンロードして設定する

ハードウェアと環境をセットアップした後、FreeRTOS をダウンロードすることができます。

FreeRTOS をダウンロードする

Nordic nRF52840-DK 用の FreeRTOS をダウンロードするには、FreeRTOS GitHub ページに移動し、リポジトリのクローンを作成します。手順については、README.md ファイルを参照してください。

> **Important**
>
> - このトピックでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。
> - *freertos* パスにスペース文字が含まれていると、構築が失敗する可能性があります。リポジトリをクローンまたはコピーするときは、作成するパスにスペース文字が含まれていないことを確認してください。

- Microsoft Windows でのファイルパスの最大長は 260 文字です。FreeRTOS のダウンロードディレクトリパスが長くなると、構築が失敗する可能性があります。
- ソースコードにはシンボリックリンクが含まれている可能性があるため、Windows を使用してアーカイブを抽出する場合は、次の操作を行う必要があります。
  - 開発者モードを有効にするか、または、
  - 管理者としてコンソールを使用します。

  この操作を行えば、Windows でアーカイブを抽出する際にシンボリックリンクを適切に作成できます。この操作を行わないと、シンボリックリンクは、そのパスがテキストとして含まれる、または空白の通常ファイルとして書き込まれます。詳細については、ブログの投稿「Symlinks in Windows 10!」を参照してください。

  Windows で Git を使用する場合は、開発者モードを有効にするか、以下を実行する必要があります。
  - 次のコマンドを使用して、`core.symlinks` を true に設定します。

    ```
    git config --global core.symlinks true
    ```
  - システムへの書き込みを行う git コマンド (git pull、git clone、git submodule update --init --recursive など) を使用する場合は、管理者としてコンソールを使用します。

プロジェクトを設定する

デモを有効にするには、 を使用するようにプロジェクトを設定する必要があります AWS IoT。と連携するようにプロジェクトを設定するには AWS IoT、デバイスを AWS IoT モノとして登録する必要があります。FreeRTOS Bluetooth Low Energy 用の AWS IoT と Amazon Cognito のセットアップすると、デバイスを用意する必要があります。

AWS IoT エンドポイントを設定するには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで [Settings] (設定) をクリックします。

   AWS IoT エンドポイントがデバイスデータエンドポイントテキストボックスに表示されます。次のようになっているはずです。*1234567890123*-ats.iot.*us-east-1*.amazonaws.comこのエンドポイントを書きとめておきます。

3. ナビゲーションペインで、[Manage] (管理)、[Things] (モノ) の順に選択します。デバイスの AWS IoT モノの名前を書き留めます。
4. AWS IoT エンドポイントと AWS IoT モノの名前を手元に用意し、IDE *freertos*/demos/include/aws_clientcredential.hで を開き、次の#define定数の値を指定します。
   - clientcredentialMQTT_BROKER_ENDPOINT *AWS IoT #######*
   - clientcredentialIOT_THING_NAME *#### AWS IoT #####*

デモを有効化するには

1. Bluetooth Low Energy GATT デモが有効になっていることを確認します。vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/iot_ble_config.h に移動して、#define IOT_BLE_ADD_CUSTOM_SERVICES (1) を define ステートメントのリストに追加します。
2. vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/aws_demo_config.h を開き、この例のように CONFIG_OTA_MQTT_BLE_TRANSPORT_DEMO_ENABLED または CONFIG_OTA_HTTP_BLE_TRANSPORT_DEMO_ENABLED を定義します。

```
/* To run a particular demo you need to define one of these.
 * Only one demo can be configured at a time
 *
 *          CONFIG_BLE_GATT_SERVER_DEMO_ENABLED
 *          CONFIG_MQTT_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_SHADOW_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_OTA_MQTT_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_OTA_HTTP_BLE_TRANSPORT_DEMO_ENABLED
 *          CONFIG_POSIX_DEMO_ENABLED
 *
 *  These defines are used in iot_demo_runner.h for demo selection */

#define CONFIG_OTA_MQTT_BLE_TRANSPORT_DEMO_ENABLED
```

3. Nordic チップに付属している RAM はメモリが非常に少ないため (250KB)、各属性のサイズと比較して大きい GATT テーブルエントリに対応するよう BLE 設定を変更する必要がある場合があります。このようにして、アプリケーションが取得するメモリの量を調整できます。これを行うには、ファイル *freertos*/vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/sdk_config.h の次の属性の定義を上書きします。

- NRF_SDH_BLE_VS_UUID_COUNT

  ベンダー固有の UUID の数。新しいベンダー固有の UUID を追加する場合は、この COUNT を 1 増やします。

- NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE

  属性テーブルのサイズ (バイト単位)。サイズは 4 の倍数にする必要があります。この値は、属性テーブル専用に設定したメモリ量 (特性サイズを含む) を示すため、プロジェクトごとに異なります。属性テーブルのサイズを超えると、NRF_ERROR_NO_MEM エラーが発生します。通常、NRF_SDH_BLE_GATTS_ATTR_TAB_SIZE を変更する場合は、RAM 設定を再構成する必要もあります。

(テストの場合、ファイルの場所は *freertos*/vendors/nordic/boards/nrf52840-dk/aws_tests/config_files/sdk_config.hです。)

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS をダウンロードし、デモプロジェクトを設定したので、ボードでデモプロジェクトを構築して実行する準備が整いました。

> **Important**
>
> このボードでデモを実行するのが初めての場合は、デモの実行前にブートローダーをボードにフラッシュする必要があります。
> ブートローダーを構築してフラッシュするには、以下の手順に従います。ただし、projects/nordic/nrf52840-dk/ses/aws_demos/aws_demos.emProject プロジェクトファイルを使用するのではなく、projects/nordic/nrf52840-dk/ses/aws_demos/bootloader/bootloader.emProject を使用します。

Segger Embedded Studio から FreeRTOS Bluetooth Low Energy デモを構築して実行するには

1. Segger Embedded Studio を開きます。上部のメニューから [File] (ファイル) を選択し、次に [Open Solution] (ソリューションを開く) を選択してプロジェクトファイル projects/nordic/nrf52840-dk/ses/aws_demos/aws_demos.emProject に移動します。

2. Segger Embedded Studio ターミナルエミュレータを使用している場合は、トップメニューから [Tools] (ツール) を選択後、[Terminal Emulator] (ターミナルエミュレータ)、[Terminal Emulator] (ターミナルエミュレータ) の順に選択して、シリアル接続の情報を表示します。

   別のターミナルツールを使用している場合は、シリアル接続の出力用にそのツールをモニタリングできます。

3. [Project Explorer] (プロジェクトエクスプローラ) の `aws_demos` デモプロジェクトを右クリックし、[Build] (構築) を選択します。

   > **Note**
   >
   > Segger Embedded Studio を初めて使用する場合は、「No license for commercial use (商用使用のためのライセンスがありません)」という警告が表示されることがあります。Segger Embedded Studio は、Nordic Semiconductor のデバイス用であれば無料で使用できます。無料ライセンスをリクエスト後、セットアップの際に [Activate Your Free License] (無料ライセンスの有効化) を選択し、手順に従います。

4. [Debug] (デバッグ)、[Go] (実行) の順に選択します。

   デモの開始後、Bluetooth Low Energy でモバイルデバイスとペアになるのを待機します。

5. MQTT over Bluetooth Low Energy Demo Application の指示に従って、FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションをモバイル MQTT プロキシとして使ってデモを完了します。

トラブルシューティング

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Nuvoton NuMaker-IoT-M487 の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Nuvoton NuMaker-IoT-M487 開発ボードの使用を開始するための手順について説明します。シリーズマイクロコントローラーには、RJ45 イーサネットおよび Wi-Fi モジュールが組み込まれています。Nuvoton NuMaker-IoT-M487 をお持ちでない場合は、AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

開始する前に、開発ボードを AWS クラウドに接続するように AWS IoT と FreeRTOS ソフトウェアを設定する必要があります。手順については、最初のステップ を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

概要

このチュートリアルでは次のステップを説明します。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

開発環境をセットアップする

Keil MDK Nuvoton エディションは、Nuvoton M487 ボード用のアプリケーションを開発およびデバッグするために設計されています。Keil MDK v5 Essential、Plus、または Pro バージョンは Nuvoton M487 (Cortex-M4 コア) MCU でも動作します。Keil MDK Nuvoton エディションは、Nuvoton Cortex-M4 シリーズ MCU の割引価格でダウンロードできます。Keil MDK は Windows でのみサポートされています。

NuMaker-IoT-M487 用の開発ツールをインストールするには

1. Keil MDK ウェブサイトから Keil MDK Nuvoton Edition をダウンロードします。
2. ライセンスを使用して Keil MDK をホストマシンにインストールします。Keil MDK には、Keil µVision IDE、C/C++ コンパイルツールチェーン、および µVision デバッガーが含まれています。

   インストール中に問題が発生した場合は、Nuvoton にお問い合わせください。
3. Nuvoton Development Tool ページにある Nu-Link_Keil_Driver_V3.06.7215r (または最新バージョン) をインストールします。

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモプロジェクトを構築するには

1. Keil µVision IDE を開きます。
2. [File] (ファイル) メニューで、[Open] (開く) を選択します。[Open file] (ファイルを開く) ダイアログボックスで、ファイルタイプセレクタが [Project Files] (プロジェクトファイル) に設定されていることを確認します。
3. 構築する Wi-Fi またはイーサネットデモプロジェクトを選択します。
   - Wi-Fi デモプロジェクトを開くには、*freertos*\projects\nuvoton\numaker_iot_m487_wifi\uvision\aws_demos ディレクトリでターゲットプロジェクト aws_demos.uvproj を選択します。
   - イーサネットデモプロジェクトを開くには、*freertos*\projects\nuvoton\numaker_iot_m487_wifi\uvision\aws_demos_eth ディレクトリでターゲットプロジェクト aws_demos_eth.uvproj を選択します。
4. ボードをフラッシュするための設定が正しいことを確認するには、IDE で aws_demo プロジェクトを右クリックし、[Options] (オプション) を選択します。(詳細については、トラブルシューティング をご覧ください。)
5. [Utilities] (ユーティリティ) タブで、[Use Target Driver for Flash Programming] (フラッシュプログラミング用のターゲットドライバーを使用する) が選択されており、[Nuvoton Nu-Link Debugger] (Nuvoton Nu-Link デバッガー) がターゲットドライバーとして設定されていることを確認します。
6. [Debug] (デバッグ) タブで、[Nuvoton Nu-Link Debugger] (Nuvoton Nu-Link デバッガー) の横にある [Settings] (設定) を選択します。
7. [Chip Type] (チップのタイプ) が [M480] に設定されていることを確認します。
8. Keil µVision IDE [Project] (プロジェクト) ナビゲーションペインで、aws_demos プロジェクトを選択します。[Project] (プロジェクト) メニューで [Build Target] (ターゲットを構築) を選択します。

AWS IoT コンソールで MQTT クライアントを使用して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。

2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

FreeRTOS デモプロジェクトを実行するには

1. Numaker-IoT-M487 ボードをホストマシン (コンピュータ) に接続します。
2. プロジェクトを再構築します。
3. Keil µVision IDE の [Flash] (フラッシュ) メニューで、[Download] (ダウンロード) を選択します。
4. [Debug] (デバッグ) メニューで、[Start/Stop Debug Session] (デバッグセッションの開始/停止) を選択します。
5. デバッガーが `main()` のブレークポイントで停止したら、[Run] (実行) メニューを開き、[Run (F5)] (実行 (F5)) を選択します。

   AWS IoT コンソールの MQTT クライアントに、デバイスによって送信された MQTT メッセージが表示されます。

FreeRTOS で CMake を使用する

CMake を使用して、サードパーティーのコードエディタやデバッグツールを使用して開発した FreeRTOS デモアプリケーションやアプリケーションを構築して実行することもできます。

CMake ビルドシステムがインストールされていることを確認します。FreeRTOS で CMake を使用する の指示に従った後に、このセクションのステップに従います。

> **Note**
>
> コンパイラ (Keil) の場所へのパスが Path システム変数 (`C:\Keil_v5\ARM\ARMCC\bin` など) にあることを確認します。

AWS IoT コンソールで MQTT クライアントを使用して、デバイスが AWS クラウドに送信するメッセージをモニタリングすることもできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

ソースファイルからビルドファイルを生成し、デモプロジェクトを実行するには

1. ホストマシンで、コマンドプロンプトを開き、*freertos* フォルダに移動します。
2. 生成されたビルドファイルを格納するフォルダを作成します。このフォルダは、*BUILD_FOLDER* と呼びます。
3. Wi-Fi またはイーサネットデモのビルドファイルを生成します。

   - Wi-Fi の場合:

     FreeRTOS デモプロジェクトのソースファイルが格納されているディレクトリに移動します。次に、以下のコマンドを実行してビルドファイルを生成します。

     ```
     cmake -DVENDOR=nuvoton -DBOARD=numaker_iot_m487_wifi -DCOMPILER=arm-keil -S . -
     B BUILD_FOLDER -G Ninja
     ```

   - イーサネットの場合:

     FreeRTOS デモプロジェクトのソースファイルが格納されているディレクトリに移動します。次に、以下のコマンドを実行してビルドファイルを生成します。

     ```
     cmake -DVENDOR=nuvoton -DBOARD=numaker_iot_m487_wifi -DCOMPILER=arm-keil -
     DAFR_ENABLE_ETH=1 -S . -B BUILD_FOLDER -G Ninja
     ```

4. 以下のコマンドを実行して、M487 にフラッシュするバイナリを生成します。

   ```
   cmake --build BUILD_FOLDER
   ```

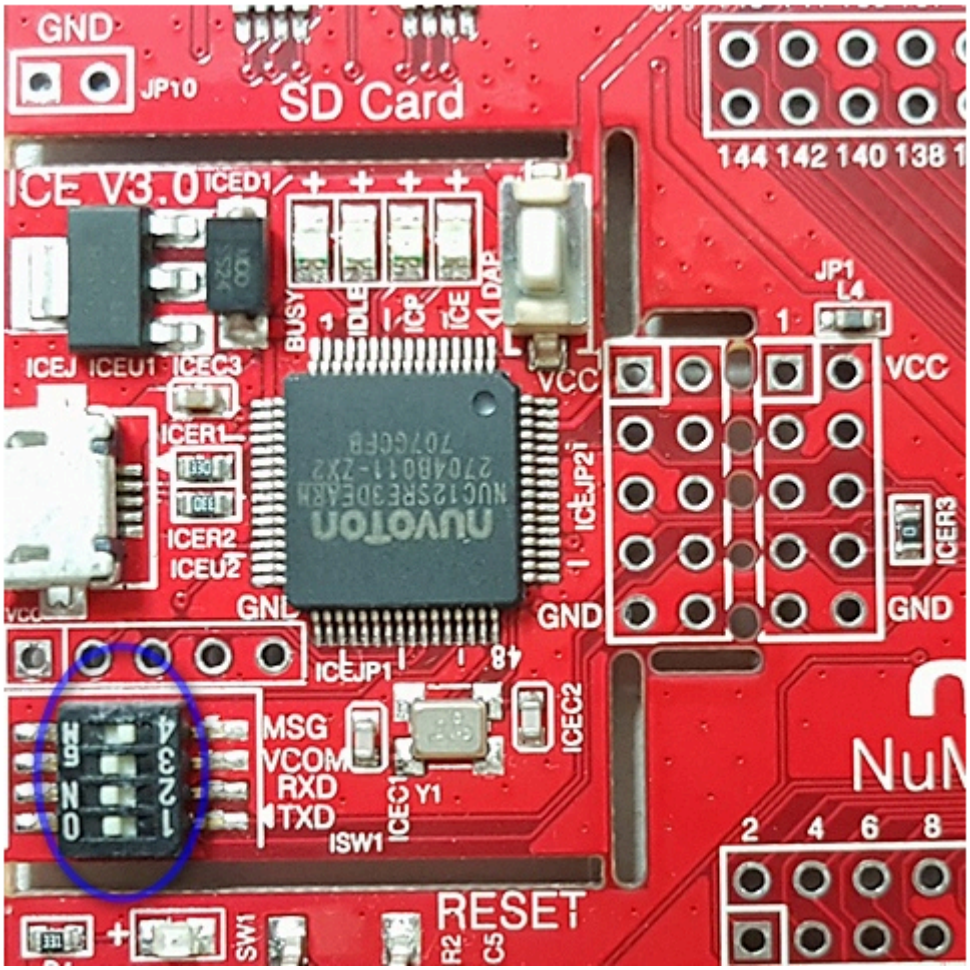
   この時点で、バイナリファイル aws_demos.bin は *BUILD_FOLDER*/vendors/Nuvoton/boards/numaker_iot_m487_wifi フォルダにある必要があります。

5. ボードをフラッシュモードに設定するには、MSG スイッチ (ICE の ISW1 の 4 番) がオンになっていることを確認します。ボードに接続すると、ウィンドウ (およびドライブ) が割り当てられます。(「トラブルシューティング」を参照してください。)
6. ターミナルエミュレータを開き、UART 経由でメッセージを表示します。ターミナルエミュレーターをインストールする の指示に従います。
7. 生成されたバイナリをデバイスにコピーして、デモプロジェクトを実行します。

   MQTT クライアントで AWS IoT MQTT トピックにサブスクライブした場合、 AWS IoT コンソールにデバイスから送信された MQTT メッセージが表示されます。

トラブルシューティング

- Windows がデバイス VCOM を認識できない場合は、Nu-Link USB Driver v1.6 から NuMaker Windows シリアルポートドライバーをインストールします。
- Nu-Link 経由でデバイスを Keil MDK (IDE) に接続する場合は、次に示すように MSG スイッチ (ICE 上の ISW1 の 4 番) がオフになっていることを確認します。



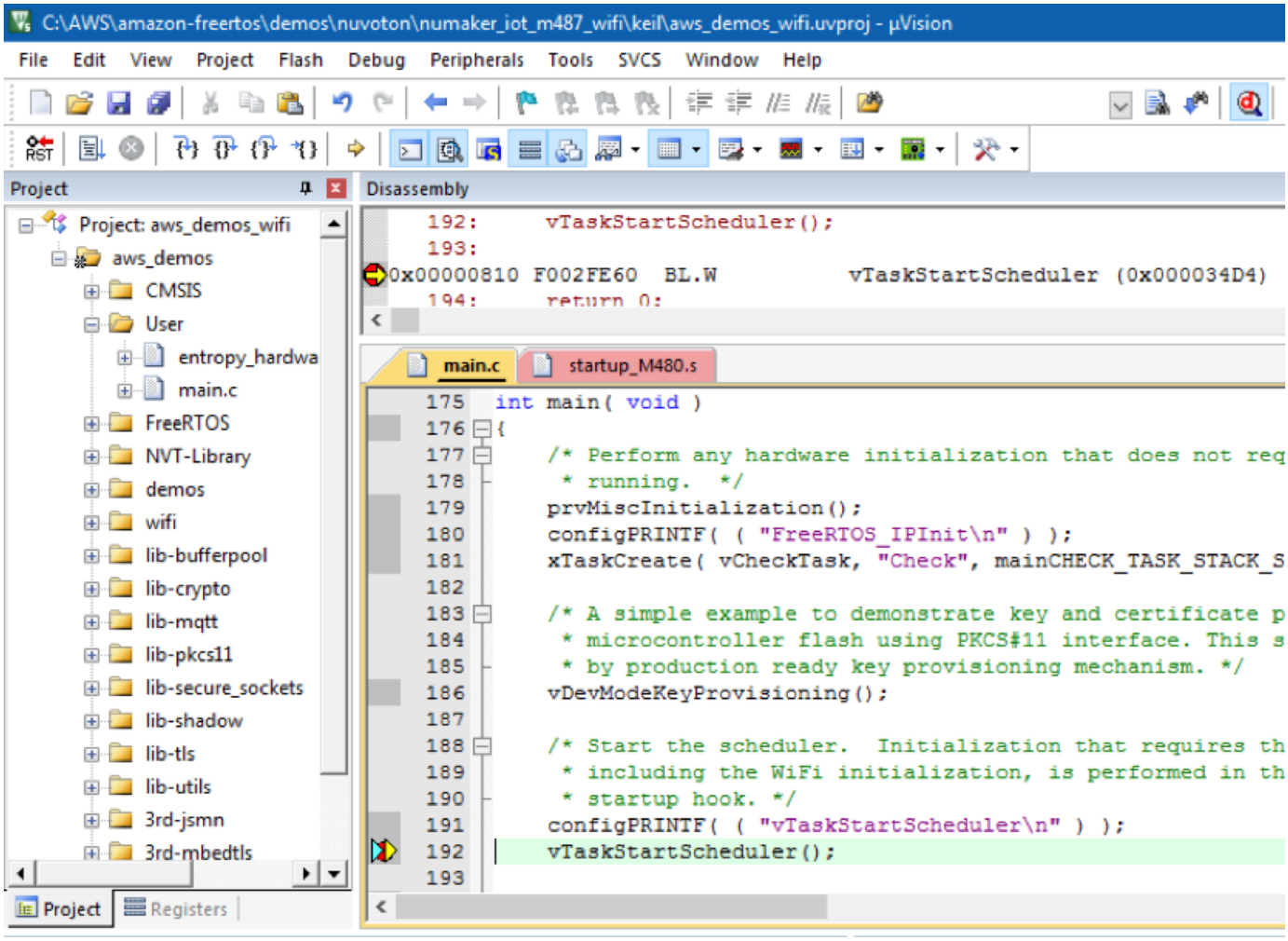
開発環境の設定やボードへの接続で問題が発生した場合は、Nuvoton にお問い合わせください。

Keil µVision での FreeRTOS プロジェクトのデバッグ

Keil µVision でデバッグセッションを開始するには

1. Keil µVision を開きます。
2. 手順に従って、FreeRTOS デモプロジェクトを構築して実行する で FreeRTOS デモプロジェクトを構築します。
3. [Debug] (デバッグ) メニューで、[Start/Stop Debug Session] (デバッグセッションの開始/停止) を選択します。

   デバッグセッションを開始すると、[Call Stack+Locals] (スタックの呼び出し + ローカル) ウィンドウが表示されます。µVision はデモをボードに点滅させ、デモを実行し、`main()` 関数の先頭で停止します。
4. プロジェクトのソースコードのブレークポイントを設定し、コードを実行します。プロジェクトは次のようになります。

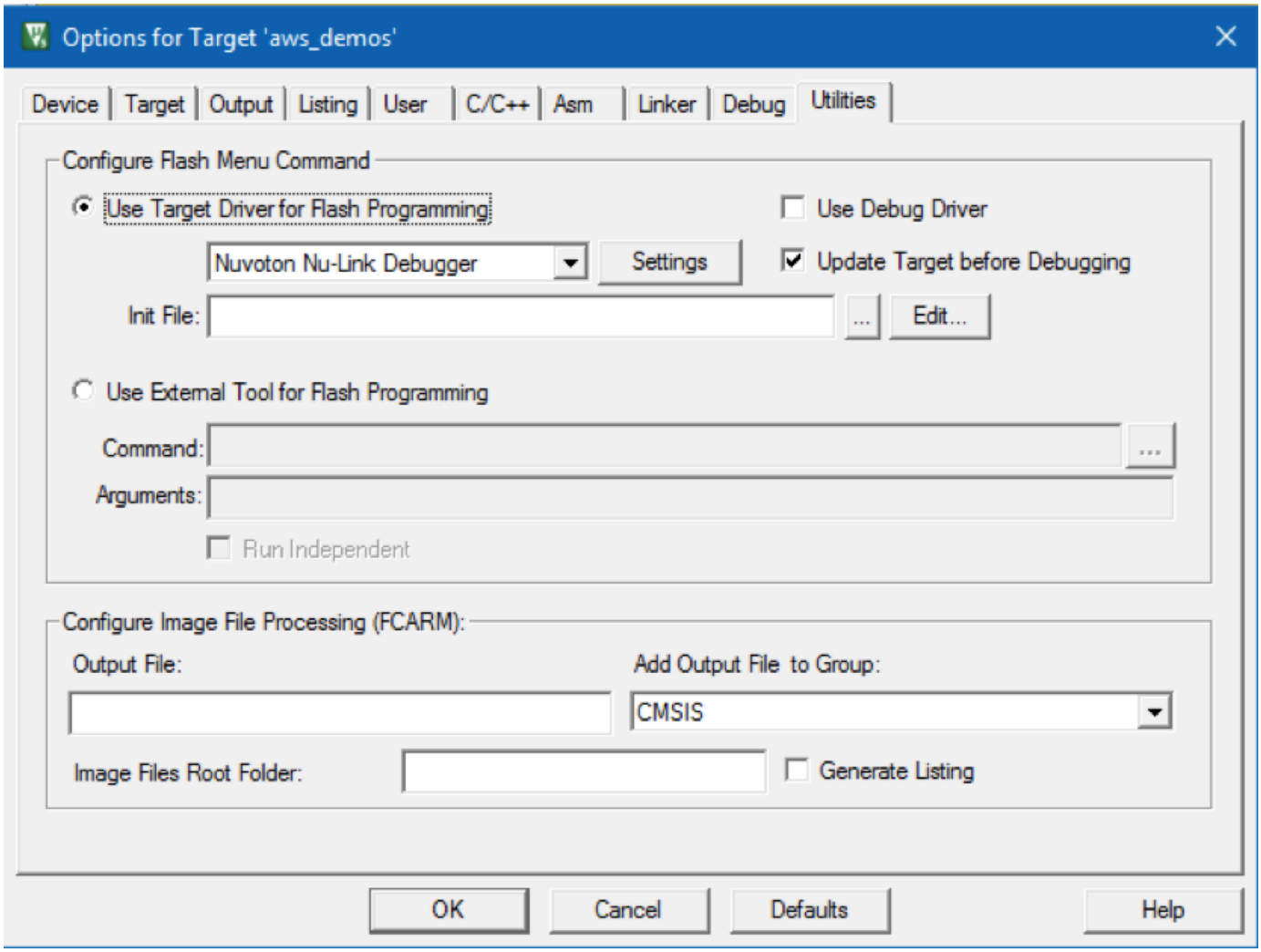


µVision デバッグ設定のトラブルシューティング

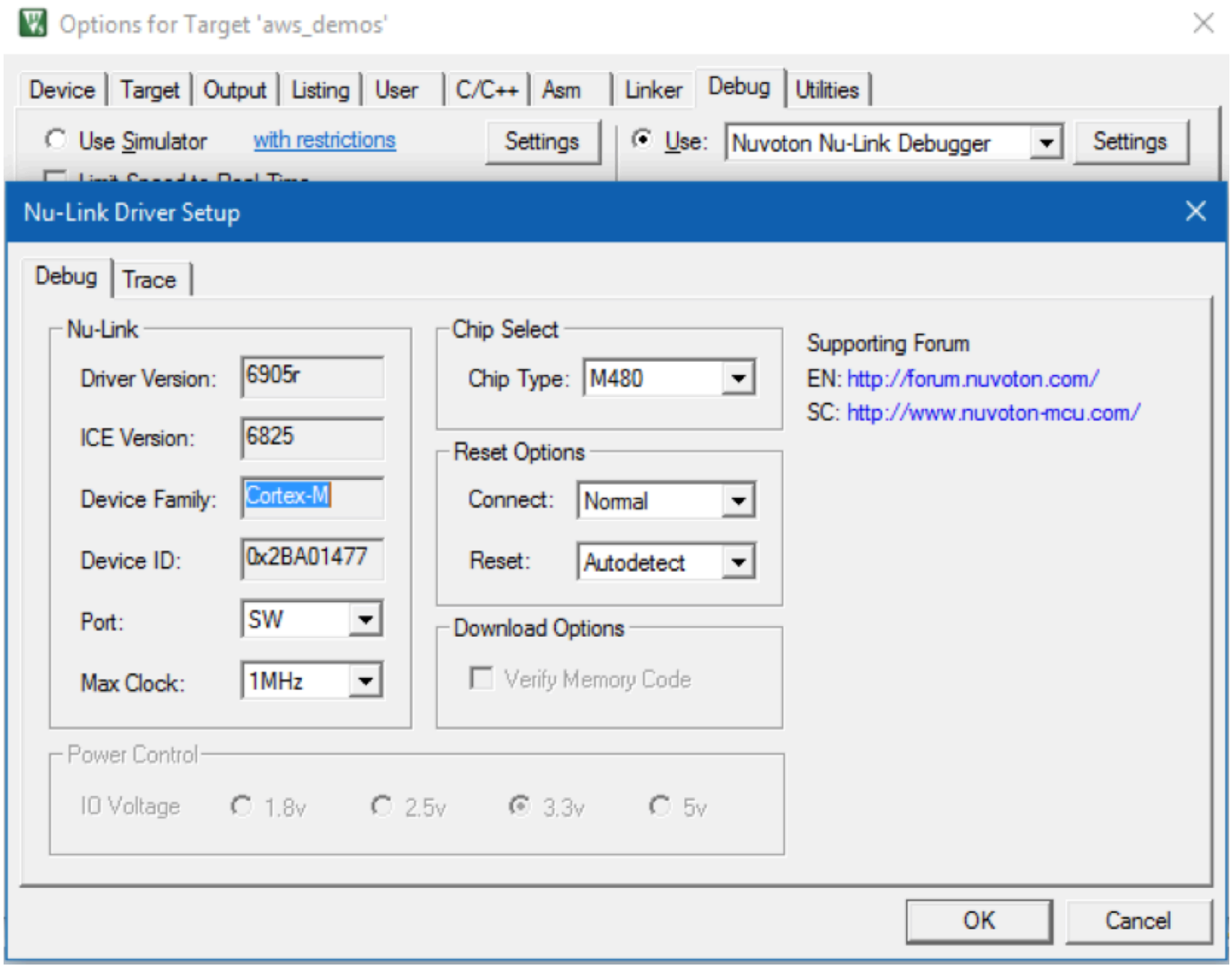
アプリケーションのデバッグ中に問題が発生した場合は、Keil µVision でデバッグ設定が正しく設定されていることを確認してください。

µVision デバッグ設定が正しいことを確認するには

1. Keil µVision を開きます。
2. IDE で aws_demo プロジェクトを右クリックし、[Options] (オプション) を選択します。
3. [Utilities] (ユーティリティ) タブで、[Use Target Driver for Flash Programming] (フラッシュプログラミング用のターゲットドライバーを使用する) が選択されており、[Nuvoton Nu-Link Debugger] (Nuvoton Nu-Link デバッガー) がターゲットドライバーとして設定されていることを確認します。



4. [Debug] (デバッグ) タブで、[Nuvoton Nu-Link Debugger] (Nuvoton Nu-Link デバッガー) の横にある [Settings] (設定) を選択します。



5. [Chip Type] (チップのタイプ) が [M480] に設定されていることを確認します。

NXP LPC54018 IoT モジュールの開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、NXP LPC54018 IoT モジュールの使用を開始するための手順について説明します。NXP LPC54018 IoT モジュールがない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。USB ケーブルを使用して NXP LPC54018 IoT モジュールをコンピュータに接続します。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. ボードをホストマシンに接続します。
2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

NXP ハードウェアを設定する

NXP LPC 54018 を設定するには

- コンピュータを LPC 54018 NXP の USB ポートに接続します。

JTAG デバッガーをセットアップするには

NXP LPC54018 ボード上で実行されるコードを起動してデバッグするには、JTAG デバッガーが必要です。FreeRTOS は OM40006 IoT モジュールを使用してテストされました。サポートされるデバッガーの詳細については、OM40007 LPC54018 IoT モジュール製品ページの NXP LPC54018 IoT モジュールのユーザーマニュアルを参照してください。

1. OM40006 IoT モジュールデバッガーを使用している場合、コンバーターケーブルを使用して 20 ピンコネクタをデバッガーから NXP IoT モジュールの 10 ピンコネクタに接続します。
2. ミニ USB to USB ケーブルを使用して、コンピュータの USB ポートに NXP LPC54018 と OM40006 IoT モジュールデバッガーを接続します。

開発環境をセットアップする

FreeRTOS は、NXP LPC54018 IoT モジュール用の 2 つの IDE (IAR Embedded Workbench と MCUXpresso) をサポートしています。

開始する前に、これらのいずれかの IDE をインストールします。

IAR Embedded Workbench for ARM をインストールするには

1. 「IAR Embed Workbench for ARM」を参照し、ソフトウェアをダウンロードします。

   > **Note**
   >
   > IAR Embedded Workbench for ARM には、Microsoft Windows が必要です。

2. インストーラを実行し、プロンプトに従います。
3. [License Wizard] (ライセンスウィザード) で [Register with IAR Systems to get an evaluation license] (IAR Systems に登録して評価ライセンスを取得する) を選択します。
4. デモを実行する前に、デバイスにブートローダを置きます。

NXP から MCUXpresso をインストールするには

1. NXP から MCUXpresso インストーラをダウンロードしてインストールします。

   > **Note**
   >
   > バージョン 10.3.x 以降がサポートされます。

2. [MCUXpresso SDK] を参照して、[Build your SDK] (SDK を構築) を選択します。

   > **Note**
   >
   > バージョン 2.5 以降がサポートされています。

3. [Select Development Board] (開発ボードの選択) を選択します。
4. [Select Development Board] (開発ボードの選択) の [Search by Name] (名前で検索) に、「**`LPC54018-IoT-Module`**」と入力し ます。
5. [Boards] (ボード) で、[LPC54018-IoT-Module] を選択します。

6. ハードウェアの詳細を確認してから、[Build MCUXepresso SDK] (MCUXepresso SDK の構築) を選択します。
7. IDE MCUXpresso を使用している SDK for Windows はすでに組み込まれています。[Download SDK] (SDK のダウンロード) を選択します。別のオペレーティングシステムを使用している場合は、[Host OS] (ホスト OS) からオペレーティングシステムを選択し、[Download SDK] (SDK のダウンロード) を選択します。
8. MCUXpresso IDE を起動し、[Installed SDK] (インストールされている SDK) タブを選択します。
9. [Installed SDK] (インストールされている SDK) ウィンドウに、ダウンロードした SDK アーカイブファイルをドラッグアンドドロップします。

インストール中に問題が発生した場合は、「NXP サポート」または「NXP 開発者用リソース」を参照してください。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモを IDE にインポートする

FreeRTOS サンプルコードを IAR Embedded Workbench IDE にインポートするには

1. IAR Embedded Workbench を開き、[File] (ファイル) メニューから、[Open Workspace] (WorkSpace を開く) を選択します。

2. [search-directory] テキストボックスに「`projects/nxp/lpc54018iotmodule/iar/aws_demos`」と入力し、[aws_demos.eww] を選択します。
3. [Project] (プロジェクト) メニューから [Rebuild All] (すべて再構築) を選択します。

FreeRTOS サンプルコードを MCUXpresso IDE にインポートするには

1. MCUXpresso を開き、[File] (ファイル) メニューから [Open Projects From File System] (ファイルシステムからプロジェクトを開く) を選択します。
2. [Directory] (ディレクトリ) テキストボックスで「`projects/nxp/lpc54018iotmodule/mcuxpresso/aws_demos`」と入力し、[Finish] (終了) を選択します。
3. [Project] (プロジェクト) メニューから [Build All] (すべて構築) を選択します。

FreeRTOS デモプロジェクトを実行する

IAR Embedded Workbench IDE で FreeRTOS デモプロジェクトを実行するには

1. IDE で、[Project] (プロジェクト) メニューから [Make] を選択します。
2. [Project] (プロジェクト) メニューから、[Download and Debug] (ダウンロードとデバッグ) を選択します。
3. [Debug] (デバッグ) メニューから [Start Debugging] (デバッグの開始) を選択します。
4. デバッガーが `main` のブレークポイントで停止したら、[Debug] (デバッグ) メニューから [Go] (実行) を選択します。

> **Note**
>
> [J-Link Device Selection] (J-Link デバイス選択) ダイアログボックスが表示された場合は、[OK] を選択して続行します。[Target Device Settings] (ターゲットデバイスの設定) ダイアログボックスで [Unspecified] (指定しない)、[Cortex-M4]、[OK] の順に選択します。これを行う必要があるのは 1 度だけです。

MCUxpresso IDE で FreeRTOS デモプロジェクトを実行するには

1. IDE で、[Project] (プロジェクト) メニューから [Build] (構築) を選択します。
2. デバッグを初めて使用する場合は、[Debug] (デバッグ) ツールバーで `aws_demos` プロジェクトを選択し、青いデバッグボタンを選択します。

3. 検出されたデバッグプローブがすべて表示されます。使用するプローブを選択し、[OK] を選択してデバッグを開始します。

> **Note**
>
> デバッガーが `main()` のブレークポイントで停止したら、デバッグ再起動ボタン
>
> 
>
> を 1 回押してデバッグセッションをリセットします。(これは、NXP54018-IoT-Module 用の MCUXpresso デバッガーのバグのために必要です。)

4. デバッガーが `main()` のブレークポイントで停止したら、[Debug] (デバッグ) メニューから [Go] (実行) を選択します。

トラブルシューティング

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Renesas Starter Kit+ for RX65N-2MB の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Renesas Starter Kit+ for RX65N-2MB の使用を開始するための手順について説明します。Renesas RSK+ for RX65N-2MB がない場合は、 AWS Partner Device Catalog にアクセスして、当社のパートナーから購入してください。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. ボードをホストマシンに接続します。
2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

Renesas ハードウェアを設定する

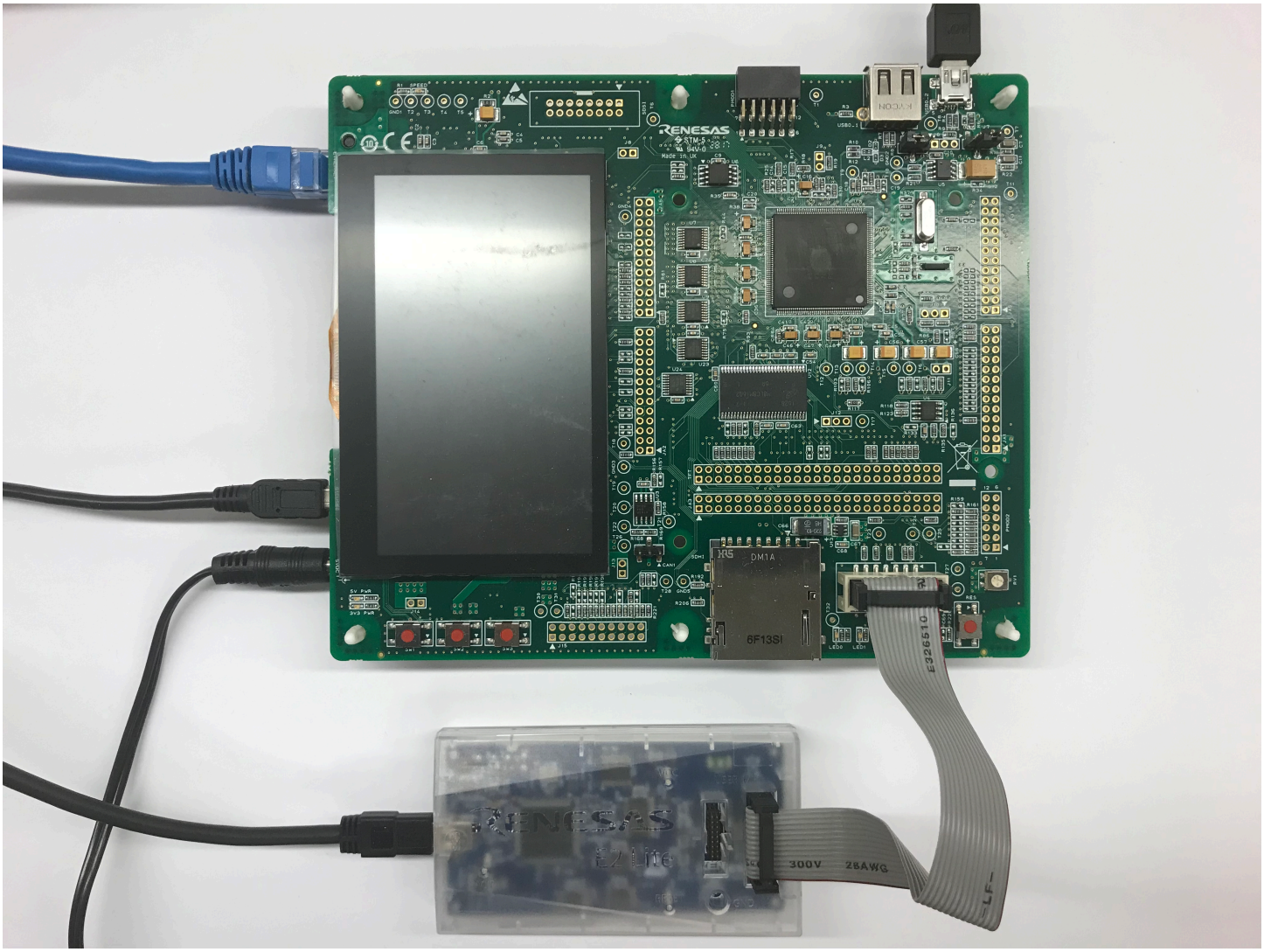
RSK+ for RX65N-2MB を設定するには

1. 正の +5V 電源アダプタを RSK+ for RX65N-2MB の PWR コネクタに接続します。
2. RSK+ for RX65N-2MB の USB2.0 FS ポートにコンピュータを接続します。
3. RSK+ for RX65N-2MB の USB-to-serial ポートにコンピュータを接続します。
4. ルーターまたはインターネットに接続されたイーサネットポートを、RSK+ for RX65N-2MB のイーサネットポートに接続します。

E2 Lite デバッガーモジュールをセットアップするには

1. 14 ピンリボンケーブルを使用して、E2 Lite デバッガーモジュールを RSK+ for RX65N-2MB の「E1/E2 Lite」ポートに接続します。
2. USB ケーブルを使用して、E2 Lite デバッガーモジュールをホストマシンに接続します。E2 Lite デバッガーをボードとコンピュータの両方に接続している場合は、デバッガーの緑色の「ACT」LED が点滅します。
3. デバッガーをホストマシンと RSK+ for RX65N-2MB に接続したら、E2 Lite デバッガーのドライバーのインストールが開始されます。

   ドライバーをインストールするには管理者権限が必要であることに注意してください。



開発環境をセットアップする

RSK+ for RX65N-2MB の FreeRTOS 設定をセットアップするには、Renesas e^{2}studio IDE および CC-RX コンパイラを使用します。

> **Note**
>
> Renesas e^{2}studio IDE と CC-RX コンパイラは、Windows 7、8、および 10 オペレーティングシステムでのみサポートされています。

e²studio をダウンロードしてインストールするには

1. Renesas e²studio インストーラのダウンロードページに移動し、オフラインインストーラをダウンロードします。
2. [Renesas Login] (Renesas ログイン) ページが表示されます。

   Renesas のアカウントを持っている場合は、サインイン認証情報を入力して [ログイン] を選択します。

   アカウントを持っていない場合は、[Register now]] (今すぐ登録) を選択し、最初の登録ステップに従います。Renesas アカウントを有効にするためのリンクが記載された E メール が届きます。このリンクに従って Renesas への登録を完了してから、Renesas にログインします。
3. ログインしたら、e²studio インストーラをコンピュータにダウンロードします。
4. インストーラを開き、ステップを完了します。

詳細については、Renesas ウェブサイトの「e²studio」を参照してください。

RX ファミリー C/C++ コンパイラパッケージをダウンロードしてインストールするには

1. RX ファミリー C/C++ コンパイラパッケージのダウンロードページに移動して、V3.00.00 パッケージをダウンロードします。
2. 実行可能ファイルを開き、コンパイラをインストールします。

詳細については、Renesas ウェブサイトの RX ファミリーの C/C++ コンパイラパッケージを参照してください。

> **Note**
> コンパイラは評価版のみ無料で利用でき、60 日間有効です。61 日目に、ライセンスキーを取得する必要があります。詳細については、評価ソフトウェアツールを参照してください。

FreeRTOS サンプルを構築して実行する

デモプロジェクトを設定したので、ボード上でプロジェクトをビルドして実行する準備が整いました。

e²studio で FreeRTOS デモを構築する

e²studio でデモをインポートしてビルドするには

1. [Start] (スタート) メニューから e²studio を起動します。
2. [Select a directory as a workspace] (ディレクトリをワークスペースとして選択) ウィンドウで、作業するフォルダを参照し、[Launch] (起動) を選択します。
3. e²studio を始めて開くと、[Toolchain Registry] (ツールチェーン登録) ウィンドウが開きます。[Renesas Toolchains] (Renesas ツールチェーン) を選択し、**CC-RX v3.00.00** が選択されていることを確認します。[Register] (登録)、[OK] の順に選択します。
4. e²studio を始めて開いている場合、[Code Generator Registration] (コードジェネレーター登録) ウィンドウが表示されます。[OK] をクリックします。
5. [Code Generator COM component register] (コードジェネレーター COM コンポーネント登録) ウィンドウが表示されます。[Please restart e²studio to use Code Generator] (コードジェネレーターを使用するには e2studio を再起動してください) で [OK] を選択します。
6. [Restart e²studio] (e2studio の再起動) ウィンドウが表示されます。[OK] をクリックします。
7. e²studio が再起動します。[Select a directory as a workspace] (ディレクトリをワークスペースとして選択) ウィンドウで、[Launch] (起動) を選択します。
8. e²studio のウェルカム画面で、[Go to the e²studio workbench] (e2studio ワークベンチに移動) 矢印アイコンを選択します。
9. [Project Explorer] (プロジェクトエクスプローラー) ウィンドウを右クリックし、[Import] (インポート) を選択します。
10. インポートウィザードで、[General] (全般)、[Existing Projects into Workspace] (既存のプロジェクトを WorkSpace へ) の順に選択してから、[Next] (次へ) を選択します。
11. [Browse] (参照) を選択し、`projects/renesas/rx65n-rsk/e2studio/aws_demos` というディレクトリを見つけて、[Finish] (終了) を選択します。
12. [Project] (プロジェクト) メニューから、[Project] (プロジェクト)、[Build All] (すべて構築) を選択します。

    ビルドコンソールに、License Manager がインストールされていないことを示す警告メッセージが表示されます。CC-RX コンパイラのライセンスキーをお持ちの場合を除き、このメッセージは無視してかまいません。License Manager をインストールするには、License Manager のダウンロードページを参照してください。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS プロジェクトを実行する

e^{2}studio でプロジェクトを実行するには

1. E2 Lite デバッガーモジュールを RSK+ for RX65N-2MB に接続していることを確認します。
2. トップメニューから、[Run] (実行)、[Debug Configuration] (デバッグ設定) を選択します。
3. [Renesas GDB Hardware Debugging] (Renesas GDB ハードウェアのデバッグ) を展開し、[aws_demos HardwareDebug] を選択します。
4. [Debugger] (デバッガー) タブ、[Connection Settings] (接続設定) タブの順に選択します。接続設定が正しいことを確認します。
5. [Debug] (デバッグ) を選択し、ボードにコードをダウンロードしてデバッグを開始します。

   `e2-server-gdb.exe` についてファイアウォール警告が表示される場合があります。[Private networks, such as my home or work network] (プライベートネットワーク (ホームネットワークや社内ネットワークなど)) をチェックし、[Allow access] (アクセスの許可) を選択します。
6. e^{2}studio から、[Renesas Debug Perspective] (Renesas デバッグパースペクティブ) に変更するよう求められる場合があります。[Yes] (はい) を選択します。

   E2 Lite デバッガーの緑色の「ACT」LED が点灯します。
7. コードがボードにダウンロードされたら、[Resume] (再開) を選択し、main 関数の最初の行までコードを実行します。[Resume] (再開) をもう一度選択し、残りのコードを実行します。

Renesas からリリースされた最新のプロジェクトについては、GitHub の `amazon-freertos` リポジトリの `renesas-rx` フォークを参照してください。

トラブルシューティング

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

STMicroelectronics STM32L4 ディスカバリキット IoT ノード用の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、STMicroelectronics STM32L4 Discovery Kit IoT Node の使用を開始するための手順について説明します。STMicroelectronics STM32L4 Discovery Kit IoT ノードをまだお持ちでない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

最新の Wi-Fi ファームウェアがインストールされていることを確認してください。最新の Wi-Fi ファームウェアをダウンロードするには、「STM32L4 ディスカバリキット IoT ノード用、低電力ワイヤレス、Bluetooth Low Energy、NFC、SubGHz、Wi-Fi」を参照してください。[Binary Resources] (バイナリリソース) から [Inventek ISM 43362 Wi-Fi module firmware update (read the readme file for instructions)] (Inventek ISM 43362 Wi-Fi モジュールファームウェアの更新 (手順については readme ファイルをお読みください)) を選択します。

開始する前に、デバイスを AWS クラウドに接続するための AWS IoT FreeRTOS ダウンロードと Wi-Fi を設定する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *`freertos`* とします。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。

2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

開発環境をセットアップする

System Workbench for STM32 をインストールする

1. [OpenSTM32.org] を参照します。
2. OpenSTM32 ウェブページに登録します。System Workbench をダウンロードするには、サインインする必要があります。
3. System Workbench for STM32 インストーラを参照して、System Workbench をダウンロードおよびインストールします。

インストール中に問題が発生した場合は、System Workbench ウェブサイト にある、よくある質問を参照してください。

FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモを STM32 System Workbench にインポートする

1. STM32 System Workbench を開き、新しい WorkSpace の名前を入力します。
2. [File] (ファイル) メニューから [Import] (インポート) を選択します。[General] (全般) を展開し、[Existing Projects into Workspace] (既存のプロジェクトを WorkSpace へ) を選択してから、[Next] (次へ) を選択します。
3. [Select Root Directory] (ルートディレクトリを選択) に、`projects/st/stm32l475_discovery/ac6/aws_demos` と入力します。
4. デフォルトでは、プロジェクト `aws_demos` が選択されている必要があります。
5. プロジェクトを STM32 System Workbench にインポートするには、[Finish] (完了) を選択します。
6. [Project] (プロジェクト) メニューから [Build All] (すべて構築) を選択します。プロジェクトがエラーなしでコンパイルされているかどうかを確認します。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS デモプロジェクトを実行する

1. USB ケーブルを使用して STMicroelectronics STM32L4 ディスカバリキット IoT ノード用をコンピュータに接続します。(使用する適切な USB ポートについては、ボードに付属している製造元のドキュメントを参照してください)。
2. [Project Explorer] (プロジェクトエクスプローラー) から `aws_demos` を右クリックして [Debug As] (デバッグ方法) を選択し、[Ac6 STM32 C/C++ Application] (Ac6 STM32 C/C++ アプリケーション) を選択します。

   初めてデバッグセッションを起動した際にデバッグエラーが発生した場合は、次の手順に従います。

   1. STM32 System Workbench の [Run] (実行) メニューから、[Debug Configurations] (デバッグ設定) を選択します。
   2. [aws_demos Debug] (aws_demos デバッグ) を選択します ([Ac6 STM32 Debugging] (Ac6 STM32 デバッグ) を展開する必要がある場合があります)。
   3. [Debugger] (デバッガー) タブを選択します。
   4. [Configuration Script] (設定スクリプト) で、[Show Generator Options] (ジェネレーターオプションを表示) を選択します。
   5. [Mode Setup] (モード設定) で、[Reset Mode] (リセットモード) を [Software System Reset] (ソフトウェアシステムリセット) に設定します。[Apply] (適用) を選択し、[Debug] (デバッグ) を選択します。
3. デバッガーが `main()` のブレークポイントで停止したら、[Run] (実行) メニューから [Resume] (再開) を選択します。

FreeRTOS で CMake を使用する

FreeRTOS 開発に IDE を使用しない場合は、代わりに CMake を使用して、サードパーティのコード工ディタおよびデバッグツールを使用して開発したデモアプリケーションまたはアプリケーションを構築して実行できます。

まず、生成されたビルドファイルを格納するフォルダ (*build-folder*) を作成します。

ビルドファイルを生成するには、次のコマンドを使用します。

```
cmake -DVENDOR=st -DBOARD=stm32l475_discovery -DCOMPILER=arm-gcc -S freertos -B build-folder
```

シェルパス内に arm-none-eabi-gcc がない場合は、AFR_TOOLCHAIN_PATH CMake 変数を設定する必要もあります。例:

```
-D AFR_TOOLCHAIN_PATH=/home/user/opt/gcc-arm-none-eabi/bin
```

FreeRTOS で CMake を使用する方法の詳細については、「FreeRTOS で CMake を使用する」を参照してください。

トラブルシューティング

デモアプリケーションから次の UART 出力が表示された場合は、Wi-Fi モジュールのファームウェアを更新する必要があります。

```
[Tmr Svc] WiFi firmware version is: xxxxxxxxxxxx
[Tmr Svc] [WARN] WiFi firmware needs to be updated.
```

最新の Wi-Fi ファームウェアをダウンロードするには、「STM32L4 ディスカバリキット IoT ノード用、低電力ワイヤレス、Bluetooth Low Energy、NFC、SubGHz、Wi-Fi」を参照してください。[Binary Resources] (バイナリリソース) で、[Inventek ISM 43362 Wi-Fi module firmware update] (Inventek ISM 43362 Wi-Fi モジュールファームウェア更新) のダウンロードリンクを選択します。

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Texas Instruments CC3220SF-LAUNCHXL の開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Texas Instruments CC3220SF-LAUNCHXL の使用を開始するための手順について説明します。Texas Instruments (TI) CC3220SF-LAUNCHXL Development Kit がない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
2. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
3. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

開発環境をセットアップする

以下のステップに従って、FreeRTOS の使用を開始するための開発環境をセットアップしてください。

FreeRTOS は、TI CC3220SF-LAUNCHXL Development Kit 用の 2 つの IDE (Code Composer Studio と IAR Embedded Workbench バージョン 8.32) をサポートしていることに注意してください。どちらの IDE でも開始することができます。

Code Composer Studio をインストールする

1. TI Code Composer Studio を参照します。

2. ホストマシン (Windows、macOS、または Linux 64-bit) のプラットフォーム用に、オフラインのインストーラをダウンロードします。
3. オフラインインストーラを解凍し、実行します。プロンプトに従います。
4. [Product Families to Install] (インストールする製品ファミリー) で、[SimpleLink Wi-Fi CC32xx Wireless MCUs] を選択します。
5. 次のページで、デバッグプローブのデフォルトの設定をそのまま使用し、[Finish] (完了) を選択します。

Code Composer Studio をインストールするときに問題が発生する場合は、TI Development Tools サポート、Code Composer Studio FAQs、および CCS トラブルシューティングを参照してください。

IAR Embedded Workbench をインストールする

1. IAR Embedded Workbench for ARM のバージョン 8.32 用 Windows インストーラをダウンロードして実行します。[Debug probe drivers] (デバッグプローブドライバ) で、[TI XDS] が選択されていることを確認します。
2. インストールを終了してプログラムを起動します。[License Wizard] (ライセンスのウィザード) ページで、[Register with IAR Systems to get an evaluation license] (IAR Systems に登録して評価ライセンスを取得する)、または独自の IAR ライセンスを使用します。

SimpleLink CC3220 SDK をインストールする

SimpleLink CC3220 SDK をインストールします。SimpleLink Wi-Fi CC3220 SDK には、CC3220SF のプログラム可能な MCU 用のドライバー、40 以上のサンプルアプリケーション、またサンプルを使用するのに必要なドキュメントが含まれています。

Uniflash をインストールする

Uniflash をインストールします。CCS Uniflash は、TI MCU 上のオンチップフラッシュメモリをプログラムするために使用されるスタンドアロンのツールです。Uniflash には、GUI、コマンドライン、スクリプトインターフェイスがあります。

最新のサービスパックをインストールする

1. TI CC3220SF-LAUNCHXL の、ピンの中央のセット (位置 = 1) に SOP ジャンパーを配置し、ボードをリセットします。
2. Uniflash を開始します。CC3220SF LaunchPad ボードが、[Detected Devices] (検出されたデバイス) の下に表示されている場合は、[Start] (スタート) を選択します。ボードが検出されない場

合は、[New Configuration] (新しい設定) のボードリストから CC3220SF-LAUNCHXL を選択して、[Start Image Creator] (Image Creator の開始) を選択します。

3. [New Project] (新しいプロジェクト) を選択します。
4. [Start new project] (新しいプロジェクトを開始) ページで、プロジェクトの名前を入力します。[Device Type] (デバイスタイプ) で [CC3220SF] を選択します。[Device Mode] (デバイスモード) で、[Develop] (開発)、[Create Project] (プロジェクトの作成) の順に選択します。
5. Uniflash アプリケーションウィンドウの右側で、[Connect] (接続) を選択します 。
6. 左側の列から、[Advanced] (アドバンスド)、[Files] (ファイル)、[Service Pack] (サービスパック) の順に選択します。
7. [Browse] (参照) を選択してから、CC3220SF SimpleLink SDK をインストールした場所に移動します。このサービスパックは、`ti/simplelink_cc32xx_sdk_`*`VERSION`*`/tools/cc32xx_tools/servicepack-cc3x20/sp_`*`VERSION`*`.bin` にあります。
8. [Burn] (焼き付け)

   ()

   ボタンを選択し、次に [Program Image (Create & Program)] (プログラムイメージ (作成およびプログラム)) を選択してサービスパックをインストールします。必ず SOP ジャンパーを 0 の位置に戻してボードをリセットしてください。

Wi-Fi プロビジョニングを設定する

ボードの Wi-Fi を設定するには、次のいずれかを実行します。

- FreeRTOS デモを設定する の説明のとおり、FreeRTOS デモアプリケーションを設定します。
- Texas Instruments から SmartConfig を使用します。

FreeRTOS デモプロジェクトを構築して実行する

TI Code Composer で FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモを TI Code Composer にインポートするには

1. TI Code Composer を開き、[OK] を選択して、デフォルトの WorkSpace 名をそのまま使用します。
2. [Getting Started] (開始方法) ページで、[Import Project] (プロジェクトのインポート) を選択します。

3. [Select search-directory] (検索ディレクトリを選択) に、`projects/ti/cc3220_launchpad/ccs/aws_demos` と入力します。デフォルトでは、プロジェクト `aws_demos` が選択されている必要があります。プロジェクトを TI Code Composer にインポートするには、[Finish] (完了) を選択します。
4. [Project Explorer] (プロジェクトエクスプローラー) で、[aws_demos] をダブルクリックしてプロジェクトをアクティブにします。
5. [Project] (プロジェクト) から [Build Project] (ビルドプロジェクト) を選択して、プロジェクトがエラーや警告なしでコンパイラされていることを確認します。

TI Code Composer で FreeRTOS デモを実行するには

1. Texas Instruments CC3220SF-LAUNCHXL 上の Sense On Power (SOP) ジャンパーが 0 の位置にあることを確認してください。詳細については、SimpleLink Wi-Fi CC3x20、CC3x3x ネットワークプロセッサユーザーガイドを参照してください。
2. USB ケーブルを使用して Texas Instruments CC3220SF-LAUNCHXL をコンピュータに接続します。
3. プロジェクトエクスプローラーで、`CC3220SF.ccxml` がアクティブなターゲット設定として選択されていることを確認します。アクティブにするには、ファイルを右クリックして [Set as active target configuration] (アクティブターゲット設定として設定する) を選択します。
4. TI Code Composer で、[Run] (実行) から [Debug] (デバッグ) を選択します。
5. デバッガーが `main()` のブレークポイントで停止したら、[Run] (実行) メニューに移動して [Resume] (再開) を選択します。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

IAR Embedded Workbench で FreeRTOS デモプロジェクトを構築して実行する

FreeRTOS デモを IAR Embedded Workbench にインポートするには

1. IAR Embedded Workbench を開き、[File] (ファイル)、[Open Workspace] (WorkSpace を開く) の順に選択します。
2. `projects/ti/cc3220_launchpad/iar/aws_demos` に移動して [aws_demos.eww] を選択してから、[OK] を選択します。
3. プロジェクト名 (`aws_demos`) を右クリックし、[Make] (作成) をクリックします。

IAR Embedded Workbench で FreeRTOS デモを実行するには

1. Texas Instruments CC3220SF-LAUNCHXL 上の Sense On Power (SOP) ジャンパーが 0 の位置にあることを確認してください。詳細については、SimpleLink Wi-Fi CC3x20、CC3x3x ネットワークプロセッサユーザーガイドを参照してください。
2. USB ケーブルを使用して Texas Instruments CC3220SF-LAUNCHXL をコンピュータに接続します。
3. プロジェクトを再構築します。

   プロジェクトを再構築するには、[Project] (プロジェクト) メニューから [Make] (作成) を選択します。
4. [Project] (プロジェクト) メニューから、[Download and Debug] (ダウンロードとデバッグ) を選択します。もし、「警告: EnergyTrace の初期化に失敗しました」が表示された場合は、無視してかまいません。EnergyTrace の詳細については、「MSP EnergyTrace Technology」を参照してください。
5. デバッガーが `main()` のブレークポイントで停止したら、[Debug] (デバッグ) メニューに移動して [Go] (実行) を選択します。

FreeRTOS で CMake を使用する

FreeRTOS 開発に IDE を使用しない場合は、代わりに CMake を使用して、サードパーティのコードエディタおよびデバッグツールを使用して開発したデモアプリケーションまたはアプリケーションを構築して実行できます。

CMake で FreeRTOS デモを構築するには

1. 生成されたビルドファイルを格納するフォルダ (*build-folder*) を作成します。
2. 検索パス ($ PATH 環境変数) に、TI CGT コンパイラバイナリがあるフォルダ (`C:\ti\ccs910\ccs\tools\compiler\ti-cgt-arm_18.12.2.LTS\bin` など) が含まれていることを確認してください。

   TI ボードで TI ARM コンパイラを使用している場合は、次のコマンドを使用してソースコードからビルドファイルを生成します。

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

詳細については、「FreeRTOS で CMake を使用する」を参照してください。

トラブルシューティング

AWS IoT コンソールの MQTT クライアントにメッセージが表示されない場合は、ボードのデバッグ設定を行う必要がある場合があります。

TI ボードのデバッグ設定を設定するには

1. Code Composer の [Project Explorer] (プロジェクトエクスプローラー) で、[aws_demos] を選択します。
2. [Run] (実行) メニューから、[Debug Configurations] (デバッグ設定) を選択します。
3. ナビゲーションペインで [aws_demos] を選択します。
4. [Target] (ターゲット) タブの [Connection Options] (接続のオプション) で、[Reset the target on a connect] (接続のターゲットをリセットする) を選択します。
5. [Apply] (適用)、[Close] (終了) の順に選択します。

上記の手順が機能しない場合は、シリアルターミナルのプログラムの出力を確認します。問題の原因を示す任意のテキストが表示されます。

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Windows Device Simulator の開始方法

このチュートリアルでは、FreeRTOS Windows Device Simulator の使用を開始するための手順について説明します。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

FreeRTOS は、指定されたプラットフォーム用の FreeRTOS ライブラリとサンプルアプリケーションを含む zip ファイルとしてリリースされています。Windows マシンでサンプルを実行するには、Windows 上で動作するように移植されたライブラリとサンプルをダウンロードしてください。この一連のファイルは、FreeRTOS simulator for Windows と呼ばれます。

> **Note**
>
> このチュートリアルは Amazon EC2 Windows インスタンスでは正常に実行できません。

開発環境をセットアップする

1. 最新バージョンの Npcap をインストールします。インストール時に「WinPcap API 互換モード」を選択します。
2. Microsoft Visual Studio をインストールします。

   Visual Studio バージョン 2017 および 2019 は動作することが確認されています。Visual Studio のすべてのエディションがサポートされます (Community、Professional、または Enterprise)。

   IDE に加えて、[Desktop development with C++] (C++ によるデスクトップ開発) コンポーネントをインストールします。

   最新の Windows 10 SDK をインストールします。これは、[Desktop development with C++] (C++ によるデスクトップ開発) コンポーネントの [Optional] (オプション) セクションで選択できます。
3. アクティブな有線イーサネット接続が存在することを確認してください。
4. (オプション) CMake ベースのビルドシステムを使用して FreeRTOS プロジェクトを構築する場合は、CMake の最新バージョンをインストールします。FreeRTOS には CMake バージョン 3.13 以降が必要です。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

デバイス上でデモプロジェクトが正常に実行されると、「Hello World!」が 購読しているトピックに複数回送信されたことを確認できます。

FreeRTOS デモプロジェクトを構築して実行する

Visual Studio または CMake を使用して FreeRTOS プロジェクトを構築できます。

Visual Studio IDE を使用して FreeRTOS デモプロジェクトを構築して実行する

1. Visual Studio にプロジェクトをロードします。

   Visual Studio の [File] (ファイル) メニューから、[Open] (開ける) を選択します。[File/Solution] (ファイル/ソリューション) を選択し、`projects/pc/windows/visual_studio/aws_demos/aws_demos.sln` に移動してから [Open] (開ける) を選択します。
2. デモプロジェクトをリターゲットします。

   指定されたデモプロジェクトは、Windows SDK によって異なりますが、Windows SDK バージョンが指定されていません。デフォルトでは、IDE でマシンに存在しない SDK バージョンを使用してデモのビルドが試行されることがあります。Windows バージョンの SDK を設定するには、`aws_demos` を右クリックして [Retarget Projects] (プロジェクトをリターゲット) を選択します。これにより、[Review Solution Actions] (ソリューションのアクションを確認) ウィンドウが開きます。マシンに存在する Windows SDK バージョン (ドロップダウンの初期値で構いません) を選択して、[OK] を選択します。
3. プロジェクトを構築して実行します。

[Build] (構築) メニューから [Build Solution] (ソリューションの構築) を選択し、ソリューションがエラーや警告なしで構築されることを確認します。[Debug] (デバッグ)、[Start Debugging] (デバッグを開始) を選択し、プロジェクトを実行します。最初の実行で、ネットワークインターフェイスを選択する必要があります。

CMake を使用して FreeRTOS デモプロジェクトを構築して実行する

Windows Simulator のデモプロジェクトをビルドするには、CMake コマンドラインツールではなく、CMake GUI を使用することをお勧めします。

CMake をインストールしたら、CMake GUI を開きます。Windows では、これは、[Start] (スタート) メニューの [CMake] の下にある [CMake (cmake-gui)] にあります。

1. FreeRTOS ソースコードディレクトリを設定します。

   GUI の [Where is the source code] (ソースコードの場所) で、FreeRTOS ソースコードディレクトリ (*freertos*) を設定します。

   [Where to build the binaries] (バイナリを構築する場所) に *freertos*/build を設定します。
2. CMake プロジェクトを設定します。

   CMake GUI で [Add Entry] (エントリの追加) を選択し、[Add Cache Entry] (キャッシュエントリの追加) ウィンドウで以下の値を設定します。

   名前

   AFR_BOARD

   タイプ

   STRING

   値

   pc.windows

   説明

   (オプション)
3. [Configure] (設定) を選択します。CMake でビルドディレクトリを作成するよう求められたら、[Yes] (はい) を選択し、[Specify generator generator for this project] (このプロジェクトの

ジェネレーターを指定) でジェネレーターを選択します。ジェネレーターとして Visual Studio を使用することをお勧めしますが、Ninja もサポートされています。(Visual Studio 2019 を使用する場合、プラットフォームはデフォルト設定ではなく Win32 に設定する必要があります)。他のジェネレーターオプションは変更せずに、[完了] を選択します。

4. CMake プロジェクトを生成して開きます。

   プロジェクトを設定した後、生成されたプロジェクトで使用可能なすべてのオプションが CMake GUI に表示されます。このチュートリアルでは、オプションはデフォルト値のままで構いません。

   [Generate] (生成) を選択して Visual Studio ソリューションを作成してから、[Open Project] (プロジェクトを開く) を選択して Visual Studio でプロジェクトを開きます。

   Visual Studio で、aws_demos プロジェクトを右クリックして、[Set as StartUp Project] (スタートアッププロジェクトとして設定) を選択します。これにより、プロジェクトをビルドして実行することができます。最初の実行で、ネットワークインターフェイスを選択する必要があります。

FreeRTOS で CMake を使用する方法の詳細については、「FreeRTOS で CMake を使用する」を参照してください。

ネットワークインターフェイスを設定する

デモプロジェクトの最初の実行では、使用するネットワークインターフェイスを選択する必要があります。プログラムは、ネットワークインターフェイスをカウントします。有線イーサネットインターフェイスの番号を見つけます。出力は次のようになります:

```
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.
```

有線イーサネットインターフェイスの番号を特定した後、アプリケーションウィンドウを閉じます。上記の例では、使用する番号は 1 です。

FreeRTOSConfig.h を開いて、configNETWORK_INTERFACE_TO_USE を有線ネットワークインターフェイスに対応する番号に設定します。

> **Important**
>
> イーサネットインターフェイスのみがサポートされています。Wi-Fi はサポートされていません。

トラブルシューティング

Windows での一般的な問題のトラブルシューティング

Visual Studio でデモプロジェクトビルドしようとしたときに次のエラーが発生する可能性があります。

```
Error "The Windows SDK version X.Y was not found" when building the provided Visual
 Studio solution.
```

プロジェクトは、コンピュータに存在する Windows SDK バージョンを対象とする必要があります。

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

Xilinx Avnet MicroZed Industrial IoT キットの開始方法

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Xilinx Avnet MicroZed Industrial IoT Kit の使用を開始するための手順について説明します。Xilinx Avnet MicroZed Industrial IoT Kit がない場合は、 AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。

開始する前に、 AWS IoT と FreeRTOS ダウンロードを設定して、デバイスを AWS クラウドに接続する必要があります。手順については「最初のステップ」を参照してください。このチュートリアルでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。

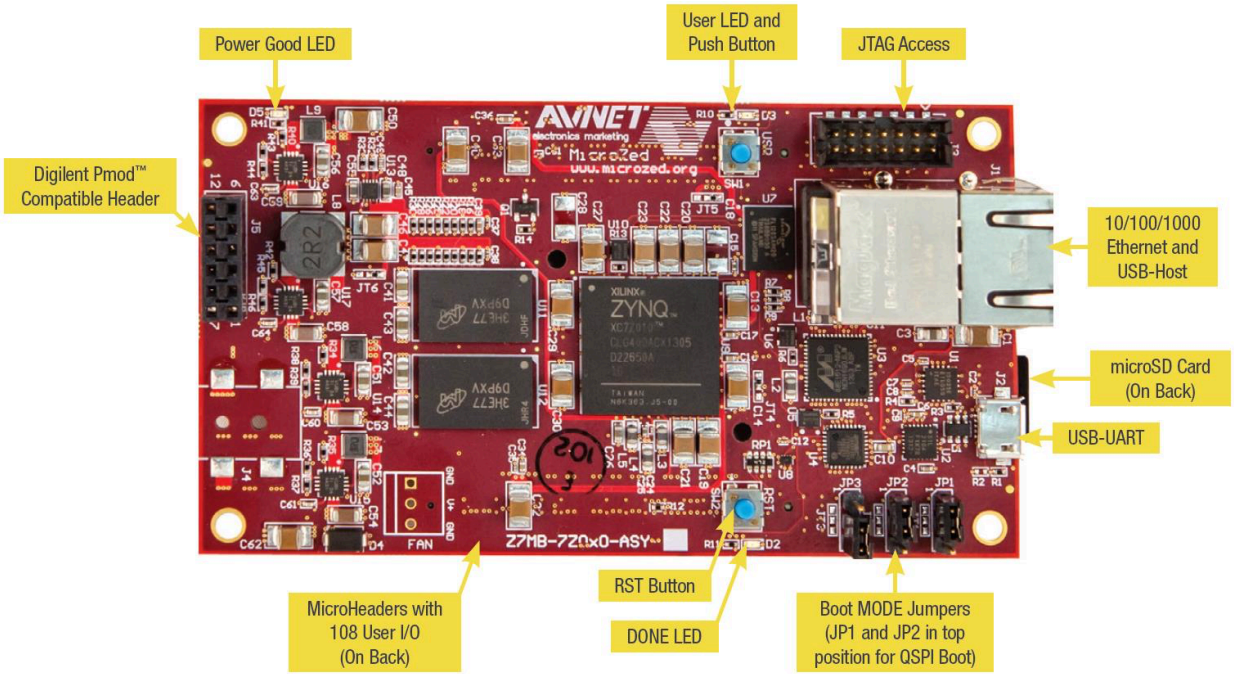
概要

このチュートリアルには、使用開始のための以下の手順が含まれています。

1. ボードをホストマシンに接続します。
2. マイクロコントローラーボード用の組み込みアプリケーションを開発およびデバッグするためのソフトウェアをホストマシンにインストールします。
3. FreeRTOS デモアプリケーションをバイナリイメージにクロスコンパイルします。
4. アプリケーションバイナリイメージをボードにロードし、アプリケーションを実行します。

MicroZed ハードウェアを設定する

以下の図は、MicroZed ハードウェアを設定するときに役立ちます。



MicroZed ボードをセットアップするには

1. MicroZed ボードの USB-UART ポートにコンピュータを接続します。
2. MicroZed ボードの JTAG Access ポートにコンピュータを接続します。
3. ルーターまたはインターネットに接続されたイーサネットポートを、MicroZed ボードのイーサネットおよび USB ホストポートに接続します。

開発環境をセットアップする

MicroZed キット用に FreeRTOS を設定するには、Xilinx ソフトウェア開発キット (XSDK) を使用する必要があります。XSDK は Windows と Linux でサポートされています。

XSDK のダウンロードとインストール

Xilinx ソフトウェアをインストールするには、無料の Xilinx アカウントが必要です。

XSDK をダウンロードするには

1. ソフトウェア開発キットスタンドアロン WebInstall クライアントのダウンロードページに移動します。
2. オペレーティングシステムに適したオプションを選択します。
3. Xilinx のサインインページが表示されます。

   Xilinx のアカウントを持っている場合は、サインイン認証情報を入力して [サインイン] を選択します。

   アカウントを持っていない場合は、[Create your account] (アカウントを作成する) を選択します。登録後、Xilinx アカウントを有効にするリンクが記載された E メール が届きます。
4. [Name and Address Verification] (名前と住所の検証) ページで、情報を入力して [Next] (次へ) を選択します。ダウンロードを開始する準備ができているはずです。
5. `Xilinx_SDK_version_os` ファイルを保存します。

XSDK をインストールするには

1. `Xilinx_SDK_version_os` ファイルを開きます。
2. [Select Edition to Install] (インストールするエディションの選択) で、[Xilinx Software Development Kit (XSDK)] (Xilinx ソフトウェア開発キット (XSDK)) を選択し、[Next] (次へ) を選択します。

3. 次のインストールウィザードのページの [Installation Options] (インストールオプション) で、[Install Cable Drivers] (ケーブルドライバーのインストール) を選択し、[Next] (次へ) を選択します。

コンピュータが MicroZed の USB-UART 接続を検出しない場合は、CP210x USB-to-UART Bridge VCP ドライバーを手動でインストールしてください。手順については、「Silicon Labs CP210x USB-to-UART インストールガイド」を参照してください。

XSDK の詳細については、Xilink のウェブサイトの「Xilinx SDK の開始方法」を参照してください。

クラウドの MQTT メッセージのモニタリング

FreeRTOS デモプロジェクトを実行する前に、 AWS IoT コンソールで MQTT クライアントを設定して、デバイスが AWS クラウドに送信するメッセージをモニタリングできます。

MQTT クライアントで AWS IoT MQTT トピックをサブスクライブするには

1. AWS IoT コンソール にサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***your-thing-name*/example/topic** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

FreeRTOS デモプロジェクトを構築して実行する

XSDK IDE で FreeRTOS デモを開く

1. ワークスペースディレクトリを *freertos*/projects/xilinx/microzed/xsdk に設定して XSDK IDE を起動します。
2. ウェルカムページを閉じます。メニューから、[Project] (プロジェクト) を選択し、[Build Automatically] (自動的に構築する) をクリアします。
3. メニューから、[File] (ファイル) を選択し、[Import] (インポート) を選択します。
4. [Select] (選択) ページで、[General] (全般) を展開し、[Existing Projects into Workspace] (既存のプロジェクトを WorkSpace へ) を選択して、[Next] (次へ) を選択します。
5. [Import Projects] (プロジェクトのインポート) ページで、[Select root directory] (ルートディレクトリの選択) を選択し、デモプロジェクトのルートディレクトリ (*freertos*/

projects/xilinx/microzed/xsdk/aws_demos) を入力します。ディレクトリを参照するには、[Browse] (参照) を選択します。

ルートディレクトリを指定すると、そのディレクトリのプロジェクトが [Import Projects] (プロジェクトのインポート) ページに表示されます。使用可能なすべてのプロジェクトはデフォルトで選択されています。

> **Note**
>
> [Import Projects] (プロジェクトのインポート) ページ (「一部のプロジェクトは既にワークスペースに存在するため、インポートできません。」) の上部に警告が表示された場合は、無視してかまいません。

6. 選択されたすべてのプロジェクトで、[Finish] (完了) を選択します。
7. プロジェクトペインに aws_bsp、fsbl、および MicroZed_hw_platform_0 プロジェクトが表示されない場合は、3 からここまでの手順を繰り返します。ただし、ルートディレクトリは *freertos*/vendors/xilinx に設定し、aws_bsp、fsbl、および MicroZed_hw_platform_0 をインポートします。
8. メニューから、[Window] (ウィンドウ) を選択し、[Preferences] (プリファレンス) を選択します。
9. ナビゲーションペインで、[Run/Debug] (実行/デバッグ) を展開して、[String Substitution] (文字列の置換) を選択し、[New] (新規) を選択します。
10. [New String Substitution Variable] (新しい文字列置換変数) の、[Name] (名前) に 「**AFR_ROOT**」と入力します。[Value] (値) で、*freertos*/projects/xilinx/microzed/xsdk/aws_demos のルートパスを入力します。[OK] を選択し、[OK] を選択して変数を保存します。[Preferences] (プリファレンス) を閉じます。

FreeRTOS デモプロジェクトを構築する

1. XSDK IDE で、メニューから、[Project] (プロジェクト) を選択し、[Clean] (クリーンアップ) を選択します。
2. [Clean] (クリーンアップ) で、オプションをデフォルト値のままにして、[OK] を選択します。XSDK はすべてのプロジェクトをクリーンアップしてビルドし、.elf ファイルを生成します。

> **Note**
>
> すべてのプロジェクトをクリーンアップせずに構築するには、[Project] (プロジェクト) を選択し、[Build All] (すべて構築) を選択します。
> 個々のプロジェクトを構築するには、構築するプロジェクトを選択し、[Project] (プロジェクト) を選択して、[Build Project] (プロジェクトを構築する) を選択します。

FreeRTOS デモプロジェクトから起動イメージを生成する

1. XSDK IDE で、[aws_demos] を右クリックして、[Create Boot Image] (起動イメージの作成) を選択します。
2. [Create Boot Image] (軌道イメージの作成) で、[Create new BIF] (新しい BIF の作成) を選択します。
3. [Output BIF file path] (BIF ファイルパスを出力する) の横で、[Browse] (参照) を選択し、*<freertos>*`/vendors/xilinx/microzed/aws_demos/aws_demos.bif` にある `aws_demos.bif` を選択します。
4. [Add] (追加) を選択します。
5. [Add new boot image partition] (新しい起動イメージパーティションの追加) で、[File path] (ファイルパス) の横の、[Browse] (参照) を選択し、`vendors/xilinx/fsbl/Debug/fsbl.elf` にある `fsbl.elf` を選択します。
6. [Partition type] (パーティションの種類) で、[bootloader] (ブートローダー) を選択し、[OK] を選択します。
7. [Create Boot Image] (軌道イメージの作成) で、[Create Image] (イメージの作成) を選択します。[Override Files] (ファイルのオーバーライド) で、[OK] を選択して既存の `aws_demos.bif` を上書きし、`projects/xilinx/microzed/xsdk/aws_demos/BOOT.bin` に `BOOT.bin` ファイルを生成します。

JTAG デバッグ

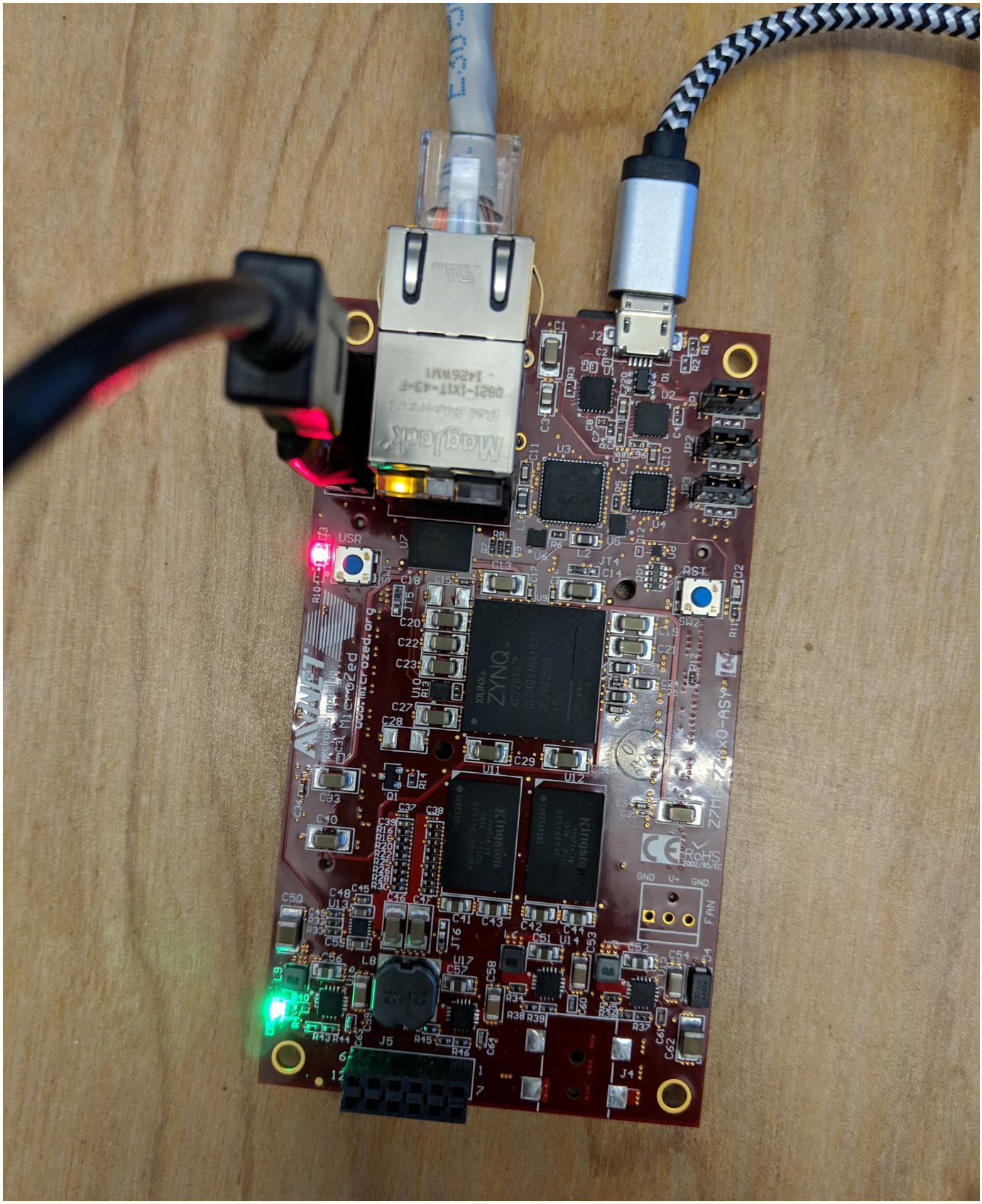
1. MicroZed ボードのブートモードジャンパーを JTAG ブートモードに設定します。



2. MicroSD カードを USB-UART ポートのすぐ下にある MicroSD カードスロットに挿入します。

> **Note**
>
> デバッグする前に、MicroSD カードにあるコンテンツを必ずバックアップしてください。

ボードは以下のようになります。


USR
RST
ZYNQ
Kingston
Kingston
RoHS
GND V+ GND
FAN

3. XSDK IDE で、[aws_demos] を右クリックして、[Debug As] (名前をつけてデバッグ) を選択し、[1 Launch on System Hardware (System Debugger)] (1 システムハードウェアで起動 (システムデバッガー)) を選択します。
4. デバッガーが `main()` のブレークポイントで停止したら、メニューから [Run] (実行) を選択し、[Resume] (再開) を選択します。

> **Note**
>
> 初めてアプリケーションを実行する際に、新しい証明書とキーのペアが不揮発性メモリにインポートされます。それ以降の実行では、イメージと `BOOT.bin` ファイルを再構築する前に、`main.c` ファイルで `vDevModeKeyProvisioning()` をコメントアウトすることができます。これにより、実行のたびに証明書とキーをストレージにコピーすることができなくなります。

MicroZed ボードを MicroSD カードまたは QSPI フラッシュから起動して、FreeRTOS デモプロジェクトを実行できます。手順については、「FreeRTOS デモプロジェクトから起動イメージを生成する」および「FreeRTOS デモプロジェクトを実行する」を参照してください。

FreeRTOS デモプロジェクトを実行する

FreeRTOS デモプロジェクトを実行するには、MicroZed ボードを MicroSD カードまたは QSPI フラッシュから起動します。

FreeRTOS デモプロジェクトを実行するために MicroZed ボードを設定するときは、MicroZed ハードウェアを設定する の図を参照してください。MicroZed ボードをコンピュータに接続していることを確認してください。

MicroSD カードから FreeRTOS プロジェクトを起動する

Xilinx MicroZed Industrial IoT キットに同梱の MicroSD カードをフォーマットします。

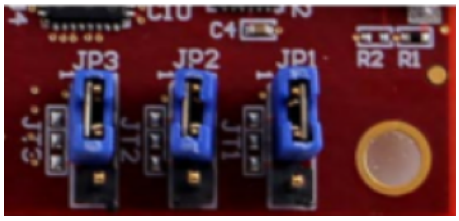
1. `BOOT.bin` ファイルを MicroSD カードにコピーします。
2. カードを USB-UART ポートのすぐ下にある MicroSD カードスロットに挿入します。
3. MicroZed ボードのブートモードジャンパーを SD ブートモードに設定します。



4. RST ボタンを押してデバイスをリセットし、アプリケーションの起動を開始します。また、USB-UART ケーブルを USB-UART ポートから抜いて、ケーブルをもう一度挿入することもできます。

QSPI フラッシュから FreeRTOS デモプロジェクトを起動する

1. MicroZed ボードのブートモードジャンパーを JTAG ブートモードに設定します。



2. コンピュータが USB-UART および JTAG アクセスポートに接続されていることを確認します。緑色の Power Good LED ライトが点灯しているはずです。
3. XSDK IDE で、メニューから、[Xilinx] を選択し、[Program Flash] (フラッシュのプログラム) を選択します。
4. [Program Flash Memory] (フラッシュメモリのプログラム) で、ハードウェアプラットフォームが自動的に入力されているはずです。[Connection] (接続) で、MicroZed ハードウェアサーバーを選択して、ホストコンピュータとボードを接続します。

> **Note**
>
> Xilinx Smart Lync JTAG ケーブルを使用している場合は、XSDK IDE にハードウェアサーバーを作成する必要があります。[New] (新規) を選択し、サーバーを定義します。

5. [Image File] (イメージファイル) で、`BOOT.bin` イメージファイルにディレクトリパスを入力します。[Browse] (参照) を選択して、代わりにファイルを参照します。
6. [Offset] (オフセット) に、**`0x0`** を入力します。
7. [FSBL File] (FSBL ファイル) で、`fsbl.elf` ファイルにディレクトリパスを入力します。[Browse] (参照) を選択して、代わりにファイルを参照します。
8. [Program] (プログラム) を選択し、ボードをプログラムします。

9. QSPI プログラミングが完了したら、USB-UART ケーブルを取り外してボードの電源を切ります。
10. MicroZed ボードのブートモードジャンパーを QSPI ブートモードに設定します。
11. カードを USB-UART ポートのすぐ下にある MicroSD カードスロットに挿入します。

> **Note**
>
> MicroSD カードにあるコンテンツを必ずバックアップしてください。

12. RST ボタンを押してデバイスをリセットし、アプリケーションの起動を開始します。また、USB-UART ケーブルを USB-UART ポートから抜いて、ケーブルをもう一度挿入することもできます。

トラブルシューティング

誤ったパスに関連するビルドエラーが発生した場合は、FreeRTOS デモプロジェクトを構築する で説明されているように、プロジェクトのクリーンアップと再ビルドを試みます。

Windows を使用している場合は、Windows XSDK IDE で文字列置換変数を設定するときにスラッシュを使用することを確認してください。

FreeRTOS の開始方法に関する一般的なトラブルシューティングについては、「トラブルシューティングの開始方法」を参照してください。

FreeRTOS での次のステップ

> **Important**
>
> このページで言及している Amazon-FreeRTOS リポジトリは非推奨です。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

ボード用の FreeRTOS デモプロジェクトを構築、フラッシュ、実行したら、FreeRTOS.org ウェブサイトにアクセスして、新しい FreeRTOS プロジェクトの作成について詳しく知ることができます。また、重要なタスクの実行、 AWS IoT サービスとの対話、ボード固有の機能 (セルラーモデムなど) のプログラミング方法を示す多くの FreeRTOS ライブラリのデモもあります。詳細については、「FreeRTOS Library Categories」ページを参照してください。

FreeRTOS.org のウェブサイトには、基本的なリアルタイムオペレーティングシステムの概念の他に、FreeRTOS カーネルに関する詳細な情報も掲載されています。詳細については、「FreeRTOS Kernel Developer Docs」ページと「FreeRTOS Kernel Secondary Docs」ページを参照してください。

FreeRTOS 無線通信経由更新

> **Note**
>
> 無線通信 (OTA) アップデートの実行に関する最新情報については、FreeRTOS ウェブサイトの「AWS IoT Over-the-Air (OTA)」を参照してください。

無線通信経由 (OTA) 更新を使用すると、フリートの 1 つまたは複数のデバイスにファームウェアをデプロイできます。OTA 更新はデバイスファームウェアの更新用に設計されていますが、 AWS IoT に登録された 1 つ以上のデバイスにファイルを送信するために使用することができます。更新を無線で送信する場合は、ファイルを受信するデバイスが、途中で改ざんされていないことを確認できるように、デジタル署名することをお勧めします。

Code Signing for AWS IoT を使用してファイルに署名したり、独自のコード署名ツールでファイルに署名したりすることができます。

OTA 更新を作成すると、OTA 更新マネージャーサービス は更新が利用可能であることをデバイスに通知する AWS IoT ジョブを作成します。OTA デモアプリケーションはデバイスで実行され、 AWS IoT ジョブの通知トピックをサブスクライブし、更新メッセージをリッスンする FreeRTOS タスクを作成します。更新が利用可能になると、OTA エージェントは、選択した設定に応じて HTTP または MQTT プロトコルを使用して、 AWS IoT へリクエストを発行し、更新を受信します。OTA エージェント は、ダウンロードしたファイルのデジタル署名をチェックし、ファイルが有効な場合はファームウェアの更新をインストールします。FreeRTOS OTA 更新デモアプリケーションを使用していない場合は、AWS IoT 無線通信経由 (OTA) ライブラリ を独自のアプリケーションに統合して、ファームウェアの更新機能を取得する必要があります。

FreeRTOS 無線通信経由更新により、次のことが可能になります。

- デプロイ前にファームウェアにデジタル署名します。
- 新しいファームウェアイメージを単一のデバイス、デバイスのグループ、またはフリート全体に展開します。
- グループに追加、リセット、または再プロビジョニングされると、デバイスにファームウェアを展開します。

- 新しいファームウェアがデバイスに導入された後、そのファームウェアの信頼性と完全性を検証します。
- デプロイの進行状況をモニタリングします。
- 失敗したデプロイをデバッグします。

OTA リソースをタグ付けする

OTA リソースの管理を支援するため、更新とストリームにはタグ形式で独自のメタデータをオプションで割り当てることができます。タグを使用すると、 AWS IoT リソースをさまざまな方法 (目的、所有者、環境など) で分類できます。これは、同じ種類のリソースが多い場合に役立ちます。リソースに割り当てたタグに基づいてリソースをすばやく特定できます。

詳細については、「AWS IoT リソースにタグを付ける」を参照してください。

OTA 更新の前提条件

無線経由 (OTA) 更新を使用するには、次の作業を行います。

- HTTP を使用した OTA 更新の前提条件 または MQTT を使用した OTA 更新の前提条件 を確認します。
- 更新を保存する Amazon S3 バケットを作成する.
- OTA 更新サービスロールを作成する.
- OTA ユーザーポリシーの作成.
- コード署名証明書の作成.
- Code Signing for を使用している場合は AWS IoT、のコード署名へのアクセスを許可する AWS IoT。
- OTA ライブラリで FreeRTOS をダウンロードする.

更新を保存する Amazon S3 バケットを作成する

OTA 更新ファイルは Amazon S3 バケットに保存されます。

Code Signing for を使用している場合 AWS IoT、コード署名ジョブの作成に使用するコマンドは、ソースバケット (署名されていないファームウェアイメージがある場所) と宛先バケット (署名されたファームウェアイメージが書き込まれる場所) を取得します。ソースと出力先に同じバケットを指定できます。元のファイルが上書きされないように、ファイル名は GUID に変更されます。

Amazon S3 バケットを作成するには

1. Amazon S3 コンソール (https://console.aws.amazon.com/s3/) にサインインします。
2. [バケットを作成] を選択します。
3. バケット名を入力します。
4. [Bucket settings for Block Public Access] (ブロックパブリックアクセスのバケット設定) で、[Block all public access] (パブリックアクセスをすべてブロック) を選択したままにして、デフォルトのアクセス許可を受け入れます。
5. [Bucket Versioning] (バケットのバージョニング) で、[Enable] (有効) を選択し、同じバケット内ですべてのバージョンを保持します。
6. [バケットを作成] を選択します。

Amazon S3 に関する詳細は、Amazon Simple Storage Service ユーザーガイドを参照してください。

OTA 更新サービスロールを作成する

OTA 更新サービスは、ユーザーに代わって OTA アップデートジョブを作成および管理するためにこのロールを引き受けます。

OTA サービスロールを作成するには

1. https://console.aws.amazon.com/iam にサインインします。
2. ナビゲーションペインで、[Roles] を選択します。
3. [Create role] (ロールの作成) を選択します。
4. [Select type of trusted entity] (信頼されたエンティティの種類を選択) の下で、[AWS Service] (AWS サービス) を選択します。
5. AWS サービスのリストから IoT を選択します。
6. [ユースケースの選択] で、[IoT] を選択します。
7. [Next: Permissions (次へ: アクセス許可)] を選択します。
8. [次へ: タグ] を選択します。
9. [Next: Review] (次へ: レビュー) を選択します。
10. ロールの名前と説明を入力し、[ロールの作成] を選択します。

IAM ロールの詳細については、IAM ロールを参照してください。

> **Important**
>
> 混乱した代理のセキュリティ問題に対処するには、「AWS IoT Core ガイド」の指示に従う必要があります。

OTA サービスロールに OTA 更新アクセス許可を追加するには

1. IAM コンソールページの検索ボックスに、ロールの名前を入力して、リストから選択します。
2. [ポリシーのアタッチ] を選択します。
3. [検索] ボックスに「AmazonFreeRTOSOTAUpdate」と入力し、フィルタリングされたポリシーの一覧で [AmazonFreeRTOSOTAUpdate] を選択してから、[ポリシーのアタッチ] を選択してポリシーをサービスロールにアタッチします。

OTA サービスロールに必要な IAM アクセス許可を追加するには

1. IAM コンソールページの検索ボックスに、ロールの名前を入力して、リストから選択します。
2. [Add inline policy] (インラインポリシーの追加) を選択します。
3. [JSON] タブを選択します。
4. 次のポリシードキュメントをコピーしてテキストボックスに貼り付けます。

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

   your_account_id を AWS アカウント ID に置き換え、*your_role_name* を OTA サービスロールの名前に置き換えてください。
5. [Review policy] (ポリシーの確認) を選択します。

6. ポリシーの名前を入力し、[Create policy] (ポリシーの作成) を選択します。

> **Note**
>
> Amazon S3 バケット名が 「afr-ota」で始まる場合、以下の手順は不要です。その場合は、AWS 管理ポリシーに必要なアクセス許可が`AmazonFreeRTOSOTAUpdate`既に含まれています。

OTA サービスロールに必要な Amazon S3 のアクセス許可を追加するには

1. IAM コンソールページの検索ボックスに、ロールの名前を入力して、リストから選択します。
2. [Add inline policy] (インラインポリシーの追加) を選択します。
3. [JSON] タブを選択します。
4. 次のポリシードキュメントをコピーしてボックスに貼り付けます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObjectVersion",
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::example-bucket/*"
            ]
        }
    ]
}
```

   このポリシーでは、Amazon S3 オブジェクトを読み取るための OTA サービスロールのアクセス許可が付与されます。*example-bucket* は、お使いのバケットの名前に置き換えてください。

5. [Review policy] (ポリシーの確認) を選択します。
6. ポリシーの名前を入力し、[Create policy] (ポリシーの作成) を選択します。

OTA ユーザーポリシーの作成

ユーザーに無線通信経由の更新を実行するアクセス許可を与える必要があります。 ユーザーには次のアクセス許可が必要です。

- ファームウェアの更新が保存されている S3 バケットにアクセスする。
- に保存されている証明書にアクセスします AWS Certificate Manager。
- AWS IoT MQTT ベースのファイル配信機能にアクセスします。
- FreeRTOS OTA 更新にアクセスする。
- アクセス AWS IoT ジョブ。
- IAM にアクセスする。
- アクセスコード署名 AWS IoT「のコード署名へのアクセスを許可する AWS IoT」を参照してください。
- FreeRTOS ハードウェアプラットフォームを一覧表示する。
- AWS IoT リソースにタグを付け、タグを解除します。

ユーザーに必要なアクセス権限を付与するには、「IAM ポリシー」を参照してください。また、AWS IoT 「ジョブの使用をユーザーとクラウドサービスに許可する」も参照してください。

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

  アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。
- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。
- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

コード署名証明書の作成

ファームウェアイメージにデジタル署名するには、コード署名証明書とプライベートキーが必要です。テストの目的で、自己署名証明書とプライベートキーを作成できます。本番稼働環境向けには、信頼された証明機関 (CA) を通じて証明書を購入してください。

プラットフォームごとに異なるタイプのコード署名証明書が必要です。次のセクションでは、各 FreeRTOS 認定プラットフォームのコード署名証明書を作成する方法について説明します。

トピック

- Texas Instruments CC3220SF-LAUNCHXL のコード署名証明書の作成
- Espressif ESP32 のコード署名証明書の作成
- Nordic nrf52840-dk のコード署名証明書の作成
- FreeRTOS Windows Simulator のコード署名証明書の作成
- カスタムハードウェアのコード署名証明書の作成

Texas Instruments CC3220SF-LAUNCHXL のコード署名証明書の作成

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

SimpleLink Wi-Fi CC3220SF Wireless Microcontroller Launchpad 開発キットは、ファームウェアコード署名用の 2 つの証明書チェーンをサポートしています。

- 本番稼働用 (証明書カタログ)

  本番稼働用証明書チェーンを使用するには、商用コード署名証明書を購入し、TI Uniflash ツールを使用してボードを本番モードに設定する必要があります。
- テストおよび開発 (証明書-プレイグラウンド)

  プレイグラウンド証明書チェーンを使用すると、自己署名コード署名証明書を使用して OTA の更新を試すことができます。

を使用して AWS Command Line Interface 、コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。詳細については、AWS Command Line Interface ユーザーガイドのAWS CLIのインストールを参照してください。

SimpleLink CC3220 SDK の最新バージョンをダウンロードしてインストールします。デフォルトでは、必要なファイルは次の場所にあります。

`C:\ti\simplelink_cc32xx_sdk_version\tools\cc32xx_tools\certificate-playground` (Windows)

`/Applications/Ti/simplelink_cc32xx_version/tools/cc32xx_tools/certificate-playground` (macOS)

SimpleLink CC3220 SDK の証明書は DER 形式です。自己署名コード署名証明書を作成するには、それらを PEM 形式に変換する必要があります。

Texas Instruments のプレイグラウンド証明書階層にリンクされ、 AWS Certificate Manager および Code Signing for AWS IoT criteria を満たすコード署名証明書を作成するには、次の手順に従います。

> **Note**
>
> コード署名証明書を作成するには、OpenSSL をマシンにインストールします。OpenSSL をインストールした後、コマンドプロンプトまたは端末環境で `openssl` が OpenSSL 実行可能ファイルに割り当てられていることを確認してください。

自己署名コード署名証明書を作成するには

1. 管理者権限でコマンドプロンプトまたは端末を開きます。
2. 作業ディレクトリで、次のテキストを使用して `cert_config.txt` という名前のファイルを作成します。*`test_signer@amazon.com`* をユーザーの E メールアドレスに置き換えてください。

```
[ req ]
prompt             = no
distinguished_name = my dn

[ my dn ]
commonName = test_signer@amazon.com
```

```
[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

3. プライベートキーと証明書署名リクエスト (CSR) を作成します。

```
openssl req -config cert_config.txt -extensions my_exts -nodes -days 365 -newkey
 rsa:2048 -keyout tisigner.key -out tisigner.csr
```

4. Texas Instruments のプレイグラウンドルート CA プライベートキーを DER 形式から PEM 形式に変換します。

   TI のプレイグラウンドルート CA プライベートキーは次の場所にあります。

   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. Texas Instruments のプレイグラウンドルート CA 証明書を DER 形式から PEM 形式に変換します。

   TI のプレイグラウンドルート CA 証明書は次の場所にあります。

   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. Texas Instruments のルート CA で CSR に署名してください。

```
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. コード署名証明書 (tisigner.crt.pem) を DER 形式に変換します。

```
openssl x509 -in tisigner.crt.pem -out tisigner.crt.der -outform DER
```

> **Note**
>
> `tisigner.crt.der` 証明書を TI 開発ボードに作成します。

8. コード署名証明書、プライベートキー、証明書チェーンを にインポートします 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
```

このコマンドは、証明書の ARN を表示します。OTA 更新ジョブを作成するときは、この ARN が必要です。

> **Note**
>
> このステップは、Code Signing for を使用してファームウェアイメージに署名することを前提 AWS IoT としています。Code Signing for の使用 AWS IoT が推奨されますが、ファームウェアイメージは手動で署名できます。

Espressif ESP32 のコード署名証明書の作成

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

Espressif ESP32 ボードは、SHA-256 と ECDSA による自己署名コード署名証明書をサポートしています。

> **Note**
>
> コード署名証明書を作成するには、OpenSSL をマシンにインストールします。OpenSSL をインストールした後、コマンドプロンプトまたは端末環境で openssl が OpenSSL 実行可能ファイルに割り当てられていることを確認してください。
> を使用して AWS Command Line Interface 、コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。のインストールの詳細については AWS CLI、「 のインストール AWS CLI」を参照してください。

1. 作業ディレクトリで、次のテキストを使用して cert_config.txt という名前のファイルを作成します。*test_signer@amazon.com* をユーザーの E メールアドレスに置き換えてください。

```
[ req ]
prompt             = no
distinguished_name = my_dn

[ my_dn ]
commonName = test_signer@amazon.com

[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

2. ECDSA のコード署名プライベートキーを作成します。

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
 ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. ECDSA のコード署名証明書を作成します。

```
openssl req -new -x509 -config cert_config.txt -extensions my_exts -nodes -days 365
 -key ecdsasigner.key -out ecdsasigner.crt
```

4. コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。

```
aws acm import-certificate --certificate fileb://ecdsasigner.crt --private-key
 fileb://ecdsasigner.key
```

このコマンドは、証明書の ARN を表示します。OTA 更新ジョブを作成するときは、この ARN が必要です。

> **Note**
>
> このステップは、Code Signing for を使用してファームウェアイメージに署名することを前提 AWS IoT としています。Code Signing for の使用 AWS IoT が推奨されますが、ファームウェアイメージは手動で署名できます。

Nordic nrf52840-dk のコード署名証明書の作成

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

Nordic nrf52840-dk は、ECDSA コード署名証明書付きの自己署名 SHA256 をサポートしています。

> **Note**
>
> コード署名証明書を作成するには、OpenSSL をマシンにインストールします。OpenSSL をインストールした後、コマンドプロンプトまたは端末環境で `openssl` が OpenSSL 実行可能ファイルに割り当てられていることを確認してください。
> を使用して AWS Command Line Interface 、コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。のインストールの詳細については AWS CLI、「 のインストール AWS CLI」を参照してください。

1. 作業ディレクトリで、次のテキストを使用して `cert_config.txt` という名前のファイルを作成します。*`test_signer@amazon.com`* をユーザーの E メールアドレスに置き換えてください。

```
[ req ]
```

```
prompt             = no
distinguished_name = my_dn

[ my_dn ]
commonName = test_signer@amazon.com

[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

2. ECDSA のコード署名プライベートキーを作成します。

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
 ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. ECDSA のコード署名証明書を作成します。

```
openssl req -new -x509 -config cert_config.txt -extensions my_exts -nodes -days 365
 -key ecdsasigner.key -out ecdsasigner.crt
```

4. コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。

```
aws acm import-certificate --certificate fileb://ecdsasigner.crt --private-key
 fileb://ecdsasigner.key
```

このコマンドは、証明書の ARN を表示します。OTA 更新ジョブを作成するときは、この ARN が必要です。

> **Note**
>
> このステップは、Code Signing for を使用してファームウェアイメージに署名することを前提 AWS IoT としています。Code Signing for の使用 AWS IoT が推奨されますが、ファームウェアイメージは手動で署名できます。

FreeRTOS Windows Simulator のコード署名証明書の作成

FreeRTOS Windows Simulator では、ECDSA P-256 キーおよび SHA-256 ハッシュを使用して OTA 更新を実行するコード署名証明書が必要です。コード署名証明書がない場合は、以下のステップを使用して証明書を作成します。

> **Note**
>
> コード署名証明書を作成するには、OpenSSL をマシンにインストールします。OpenSSL をインストールした後、コマンドプロンプトまたは端末環境で openssl が OpenSSL 実行可能ファイルに割り当てられていることを確認してください。
> を使用して AWS Command Line Interface 、コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。のインストールの詳細については AWS CLI、「 のインストール AWS CLI」を参照してください。

1. 作業ディレクトリで、次のテキストを使用して cert_config.txt という名前のファイルを作成します。*test_signer@amazon.com* をユーザーの E メールアドレスに置き換えてください。

```
[ req ]
prompt             = no
distinguished_name = my_dn

[ my_dn ]
commonName = test_signer@amazon.com

[ my_exts ]
keyUsage         = digitalSignature
extendedKeyUsage = codeSigning
```

2. ECDSA のコード署名プライベートキーを作成します。

```
openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
 ec_param_enc:named_curve -outform PEM -out ecdsasigner.key
```

3. ECDSA のコード署名証明書を作成します。

```
openssl req -new -x509 -config cert_config.txt -extensions my_exts -nodes -days 365
 -key ecdsasigner.key -out ecdsasigner.crt
```

4. コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。

```
aws acm import-certificate --certificate fileb://ecdsasigner.crt --private-key
 fileb://ecdsasigner.key
```

このコマンドは、証明書の ARN を表示します。OTA 更新ジョブを作成するときは、この ARN が必要です。

> **Note**
>
> このステップは、Code Signing for を使用してファームウェアイメージに署名することを前提 AWS IoT としています。Code Signing for の使用 AWS IoT が推奨されますが、ファームウェアイメージは手動で署名できます。

カスタムハードウェアのコード署名証明書の作成

適切なツールセットを使用して、ハードウェアの自己署名証明書とプライベートキーを作成します。

を使用して AWS Command Line Interface 、コード署名証明書、プライベートキー、証明書チェーンを にインポートします AWS Certificate Manager。のインストールの詳細については AWS CLI、「 のインストール AWS CLI」を参照してください。

コード署名証明書を作成したら、 AWS CLI を使用して ACM にインポートできます。

```
aws acm import-certificate --certificate fileb://code-sign.crt --private-key fileb://
code-sign.key
```

このコマンドの出力には、証明書の ARN が表示されます。OTA 更新ジョブを作成するときは、この ARN が必要です。

ACM では、特定のアルゴリズムとキーサイズを使用するために証明書が必要です。詳細については、「証明書をインポートする前提条件」を参照してください。ACM の詳細については、AWS Certificate Managerへの証明書のインポートを参照してください。

後でダウンロードする FreeRTOS コードの一部である `vendors/vendor/boards/board/aws_demos/config_files/ota_demo_config.h` ファイルに、コード署名証明書の内容をコピー、貼り付け、フォーマットする必要があります。

のコード署名へのアクセスを許可する AWS IoT

アクセス権限を付与するにはユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM Identity Center :

アクセス許可セットを作成します。「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」の手順に従ってください。

- IAM 内で、ID プロバイダーによって管理されているユーザー:

  ID フェデレーションのロールを作成します。詳細については「IAM ユーザーガイド」の「サードパーティー ID プロバイダー (フェデレーション) 用のロールを作成する」を参照してください。

- IAM ユーザー:
  - ユーザーが担当できるロールを作成します。手順については「IAM ユーザーガイド」の「IAM ユーザーのロールの作成」を参照してください。
  - (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。詳細については「IAM ユーザーガイド」の「ユーザー (コンソール) へのアクセス権限の追加」を参照してください。

OTA ライブラリで FreeRTOS をダウンロードする

FreeRTOS は GitHub からクローンを作成またはダウンロードできます。手順については、README.md ファイルを参照してください。

OTA デモアプリケーションの設定と実行については、「無線通信経由更新デモアプリケーション」を参照してください。

> **Important**
>
> - このトピックでは、FreeRTOS ダウンロードディレクトリへのパスを *freertos* とします。
> - *freertos* パスにスペース文字が含まれていると、構築が失敗する可能性があります。リポジトリをクローンまたはコピーするときは、作成するパスにスペース文字が含まれていないことを確認してください。
> - Microsoft Windows でのファイルパスの最大長は 260 文字です。FreeRTOS のダウンロードディレクトリパスが長くなると、構築が失敗する可能性があります。
> - ソースコードにはシンボリックリンクが含まれている可能性があるため、Windows を使用してアーカイブを抽出する場合は、次の操作を行う必要があります。
>   - 開発者モードを有効にするか、または、
>   - 管理者としてコンソールを使用します。

この操作を行えば、Windows でアーカイブを抽出する際にシンボリックリンクを適切に作成できます。この操作を行わないと、シンボリックリンクは、そのパスがテキストとして含まれる、または空白の通常ファイルとして書き込まれます。詳細については、ブログの投稿「Symlinks in Windows 10!」を参照してください。

Windows で Git を使用する場合は、開発者モードを有効にするか、以下を実行する必要があります。

- 次のコマンドを使用して、`core.symlinks` を true に設定します。

```
git config --global core.symlinks true
```

- システムへの書き込みを行う git コマンド (git pull、git clone、git submodule update --init --recursive など) を使用する場合は、管理者としてコンソールを使用します。

MQTT を使用した OTA 更新の前提条件

このセクションでは、MQTT を使用して Over-the-air (OTA) による更新を実行するための一般的な要件について説明します。

最小要件

- デバイスファームウェアに必要な FreeRTOS ライブラリ (coreMQTT エージェント、OTA 更新、その依存関係) が含まれている必要があります。
- FreeRTOS バージョン 1.4.0 以降が必要です。ただし、可能な場合は最新バージョンを使用することをお勧めします。

設定

バージョン 201912.00 以降、FreeRTOS OTA は HTTP プロトコルまたは MQTT プロトコルを使用して、 からデバイスにファームウェア更新イメージ AWS IoT を転送できます。FreeRTOS で OTA 更新を作成するときに両方のプロトコルを指定すると、各デバイスによってイメージの転送に使用するプロトコルが決定されます。詳細については「HTTP を使用した OTA 更新の前提条件」を参照してください。

デフォルトでは、`ota_config.h` での OTA プロトコルの設定は、MQTT プロトコルを使用します。

デバイス固有の設定

なし。

メモリ使用量

MQTT をデータ転送に使用する場合、MQTT 接続は制御オペレーションとデータオペレーションの間で共有されるため、追加のメモリは必要ありません。

デバイスポリシー

MQTT を使用して OTA 更新を受信する各デバイスは、 にモノとして登録する必要があり AWS IoT 、モノにはここにリストされているようなポリシーがアタッチされている必要があります。"Action" および "Resource" オブジェクトの項目の詳細については、AWS IoT Core ポリシーアクションとAWS IoT Core アクションリソースを参照してください。

```
{
    "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/*"
            ]
        }
    ]
}
```

メモ

- アクセスiot:Connect許可により、デバイスは MQTT AWS IoT 経由で に接続できます。
- AWS IoT ジョブ (.../jobs/*) のトピックに対する iot:Subscribeおよび アクセスiot:Publish許可により、接続されたデバイスはジョブ通知とジョブドキュメントを受信し、ジョブ実行の完了状態を発行できます。
- OTA ストリーム (.../streams/*) AWS IoT のトピックに対する iot:Subscribeおよび アクセスiot:Publish許可により、接続されたデバイスは OTA 更新データを取得できます AWS IoT。これらのアクセス許可は、MQTT を介してファームウェア更新を実行するために必要です。
- アクセスiot:Receive許可により、 AWS IoT Core はそれらのトピックに関するメッセージを接続されたデバイスに発行できます。このアクセス許可は、MQTT メッセージの配信ごとにチェックされます。このアクセス許可を使用して、トピックに現在サブスクライブしているクライアントへのアクセスを取り消すことができます。

HTTP を使用した OTA 更新の前提条件

このセクションでは、HTTP を使用して無線 (OTA) による更新を実行するための一般的な要件について説明します。バージョン 201912.00 以降、FreeRTOS OTA は HTTP プロトコルまたは MQTT プロトコルを使用して、 からデバイスにファームウェア更新イメージ AWS IoT を転送できます。

> **Note**
>
> - HTTP プロトコルを使用してファームウェアイメージを転送できますが、 との他のやり取りでは、ジョブ実行通知、ジョブドキュメント、実行ステータスの更新の送受信など、coreMQTT エージェントライブラリ AWS IoT Core が使用されるため、coreMQTT エージェントライブラリは依然として必要です。 coreMQTT
> - OTA 更新ジョブに MQTT プロトコルと HTTP プロトコルの両方を指定すると、各デバイスの OTA エージェントソフトウェアの設定によって、ファームウェアイメージの転送に使用するプロトコルが決定されます。OTA エージェントをデフォルトの MQTT プロトコ

ルメソッドから HTTP プロトコルに変更するには、デバイスの FreeRTOS ソースコードのコンパイルに使用するヘッダーファイルを変更します。

最小要件

- デバイスファームウェアに必要な FreeRTOS ライブラリ (coreMQTT 、HTTP、OTA エージェント、その依存関係) が含まれている必要があります。
- FreeRTOS バージョン 201912.00 以降では、HTTP 経由の OTA データ転送を有効にするために OTA プロトコルの設定を変更する必要があります。

設定

\vendors\boards*board*\aws_demos\config_files\ota_config.h ファイルにある以下の OTA プロトコルの設定を参照してください。

HTTP 経由で OTA データの転送を有効にするには

1. `configENABLED_DATA_PROTOCOLS` を `OTA_DATA_OVER_HTTP` に変更します。
2. OTA が更新されると、MQTT または HTTP のプロトコルのいずれかを使用できるように、両方のプロトコルを指定できます。`configOTA_PRIMARY_DATA_PROTOCOL` を `OTA_DATA_OVER_HTTP` に変更することで、デバイスが使用するプライマリプロトコルを HTTP に設定できます。

> **Note**
>
> HTTP は、OTA データオペレーションに対してのみサポートされます。制御オペレーションには MQTT を使用する必要があります。

デバイス固有の設定

ESP32

RAM の容量が限られているため、OTA データプロトコルとして HTTP を有効にする場合は、BLE をオフにする必要があります。vendors/espressif/boards/esp32/aws_demos/config_files/aws_iot_network_config.h ファイルで、`AWSIOT_NETWORK_TYPE_WIFI` を `configENABLED_NETWORKS` のみに変更します。

```
/**
     * @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 )
```

メモリ使用量

MQTT をデータ転送に使用する場合、MQTT 接続は制御オペレーションとデータオペレーションの間で共有されるため、追加のヒープメモリは必要ありません。ただし、HTTP 経由でデータを有効化するには、追加のヒープメモリが必要です。以下は、FreeRTOS `xPortGetFreeHeapSize` API を使用して計算された、サポートされているすべてのプラットフォームのヒープメモリの使用状況のデータです。OTA ライブラリを使用するのに十分な RAM があることを確認する必要があります。

Texas Instruments CC3220SF-LAUNCHXL

制御オペレーション (MQTT): 12 KB

データオペレーション (HTTP): 10 KB

> **Note**
>
> TI はハードウェアで SSL を行うため、使用する RAM は大幅に少なくなります。そのため、mbedtls ライブラリを使用しません。

Microchip Curiosity PIC32MZEF

制御オペレーション (MQTT): 65 KB

データオペレーション (HTTP): 43 KB

Espressif ESP32

制御オペレーション (MQTT): 65 KB

データオペレーション (HTTP): 45 KB

> **Note**
>
> ESP32 の BLE では、約 87 KB の RAM を使用します。上記のデバイス固有の設定で説明されているように、すべての設定を有効にするのに十分な RAM がありません。

Windows simulator

制御オペレーション (MQTT): 82 KB

データオペレーション (HTTP): 63 KB

Nordic nrf52840-dk

HTTP はサポートされていません。

デバイスポリシー

このポリシーでは、OTA アップデートに MQTT または HTTP を使用できます。

HTTP を使用して OTA 更新を受信する各デバイスは、 AWS IoT にモノとして登録する必要があります。また、モノには、ここに記載されているようなポリシーがアタッチされている必要があります。"Action" および "Resource" オブジェクトの項目の詳細については、AWS IoT Core ポリシーアクションとAWS IoT Core アクションリソースを参照してください。

```
{
    "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/*"
            ]
        }
    ]
}
```

メモ

- この iot:Connect アクセス許可により、デバイスが MQTT 経由で AWS IoT に接続できるようになります。
- AWS IoT ジョブ (.../jobs/*) のトピックに対する iot:Subscribeおよび アクセスiot:Publish許可により、接続されたデバイスはジョブ通知とジョブドキュメントを受信し、ジョブ実行の完了状態を発行できます。
- アクセスiot:Receive許可により AWS IoT Core 、 はそれらのトピックに関するメッセージを現在の接続デバイスに発行できます。このアクセス許可は、MQTT メッセージの配信ごとにチェックされます。このアクセス許可を使用して、トピックに現在サブスクライブしているクライアントへのアクセスを取り消すことができます。

OTA チュートリアル

このセクションには、OTA 更新を使用して FreeRTOS を実行するデバイスのファームウェアを更新するためのチュートリアルが含まれます。ファームウェアイメージに加えて、OTA 更新を使用して、任意の種類のファイルを AWS IoTに接続されたデバイスに送信できます。

AWS IoT コンソールまたは AWS CLI を使用して OTA 更新を作成できます。処理の多くを実行してくれるコンソールは、OTA を開始する一番簡単な方法です。 AWS CLI は、OTA 更新ジョブを自動

化する場合、多数のデバイスを操作する場合、または FreeRTOS に認定されていないデバイスを使用している場合に便利です。FreeRTOS の使用条件を満たすデバイスの詳細については、FreeRTOS パートナーのウェブサイトを参照してください。

OTA 更新を作成するには

1. 1 つ以上のデバイスにファームウェアの初期バージョンをデプロイします。
2. ファームウェアが正常に実行中であることを確認します。
3. ファームウェアの更新が必要な場合は、コードを変更して新しいイメージを作成します。
4. ファームウェアに手動でサインインしている場合、署名したファームウェアイメージを Amazon S3 バケットにアップロードしてください。Code Signing for を使用している場合は AWS IoT、署名されていないファームウェアイメージを Amazon S3 バケットにアップロードします。
5. OTA 更新を作成します。

OTA 更新を作成するときは、イメージ配信プロトコル (MQTT または HTTP) を指定するか、両方を指定してデバイスを選択できるようにします。デバイス上の FreeRTOS OTA エージェントは、更新されたファームウェアイメージを受信し、新しいイメージのデジタル署名、チェックサム、およびバージョン番号を検証します。ファームウェアの更新が確認されると、デバイスはリセットされ、アプリケーション定義のロジックに基づいて更新がコミットされます。デバイスが FreeRTOS を実行していない場合、デバイス上で動作する OTA エージェントを実装する必要があります。

初期ファームウェアのインストール

ファームウェアを更新するには、OTA エージェントライブラリを使用するファームウェアの初期バージョンをインストールして、OTA 更新ジョブを待機する必要があります。FreeRTOS を実行していない場合は、この手順をスキップしてください。代わりに OTA エージェント実装をデバイスにコピーする必要があります。

トピック

- Texas Instruments CC3220SF-LAUNCHXL にファームウェアの初期バージョンをインストールする
- Espressif ESP32 にファームウェアの初期バージョンをインストールする
- Nordic nRF52840 DK にファームウェアの初期バージョンをインストールする
- Windows Simulator の初期ファームウェア
- カスタムボードにファームウェアの初期バージョンをインストールする

Texas Instruments CC3220SF-LAUNCHXL にファームウェアの初期バージョンをインストールする

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

これらの手順は、Texas Instruments CC3220SF-LAUNCHXL での FreeRTOS OTA デモのダウンロード、構築、フラッシュ、実行 で説明されている、`aws_demos` プロジェクトを既に構築していることを前提としています。

1. Texas Instruments CC3220SF-LAUNCHXL の中央のピンセット (位置 = 1) に SOP ジャンパーを配置し、ボードをリセットします。
2. TI Uniflash ツールをダウンロードし、インストールします。
3. Uniflash を開始します。設定のリストから [CC3220SF-LAUNCHXL] を選択し、[Start Image Creator (Image Creator の開始)] を選択します。
4. [New Project] (新しいプロジェクト) を選択します。
5. [Start new project] (新しいプロジェクトを開始) ページで、プロジェクトの名前を入力します。[Device Type] (デバイスタイプ) で [CC3220SF] を選択します。[Device Mode (デバイスモード)] で、[Develop (開発)] を選択します。[プロジェクトを作成] を選択します。
6. ターミナルエミュレーターを切断します。
7. Uniflash アプリケーションウィンドウの右側で、[Connect] (接続) を選択します 。
8. [詳細]、[ファイル] の下で、[ユーザーファイル] を選択します。
9. [File] (ファイル) セレクタペインで、[Add File] アイコン
   
   (ファイルを追加する) を選択します。
10. `/Applications/Ti/simplelink_cc32xx_sdk_version/tools/cc32xx_tools/certificate-playground` ディレクトリに移動し、`dummy-root-ca-cert` を選択し、[Open (オープン)] を選択してから、[Write (書き込み)] を選択します。
11. [File] (ファイル) セレクタペインで、[Add File] アイコン
    
    (ファイルを追加する) を選択します。

12. コード署名証明書とプライベートキーを作成した作業ディレクトリを参照し、`tisigner.crt.der` を選択し、 [Open (オープン)] を選択してから、[Write (書き込み)] を選択します。
13. [Action (アクション)] ドロップダウンリストから [Select MCU Image (MCU イメージの選択)] を選択し、[Browse (参照)] を選択してデバイスに書き込むファームウェアイメージ ([aws_demos.bin]) を選択します。この ファイルは *`freertos`*`/vendors/ti/boards/cc3220_launchpad/aws_demos/ccs/Debug` ディレクトリにあります。開く をクリックします。

    a. ファイルダイアログボックスで、ファイル名が `mcuflashimg.bin` に設定されていることを確認します。

    b. [Vendor (ベンダー)] チェックボックスをオンにします。

    c. [File Token (ファイルトークン)] に、**`1952007250`** と入力します。

    d. [Private Key File Name (プライベートキーファイル)] で、[参照] を選択し、コード署名証明書とプライベートキーを作成した作業ディレクトリから `tisigner.key` を選択します。

    e. [Certification File Name (認定ファイル名)] で、`tisigner.crt.der`を選択します。

    f. [Write (書き込み)] を選択します。

14. 左側のペインで、[Files (ファイル)] の下にある、[Service Pack (サービスパック)] を選択します。
15. [Service Pack File Name (サービスパックファイル名)] で [Browse (参照)] を選択し、`simplelink_cc32x_sdk_`*`version`*`/tools/cc32xx_tools/servicepack-cc3x20` を参照し、`sp_3.7.0.1_2.0.0.0_2.2.0.6.bin` を選択して [Open (オープン)] を選択します。
16. 左側のペインで、[Files (ファイル)] の下にある、[Trusted Root-Certificate Catalog (信頼されたルート証明書のカタログ)] を選択します。
17. [Use default Trusted Root-Certificate Catalog (信頼されたルート証明書のカタログを使用)] チェックボックスをクリアします。
18. [ソースファイル] の下で、[参照] を選択し、[simplelink_cc32xx_sdk_***`version`***/tools/cc32xx_tools/certificate-playground/certcatalogPlayGround20160911.lst] を選択します。次に [開く] を選択します。
19. [Signature Source File (署名ソースファイル)] の下で、[参照] を選択し、[simplelink_cc32xx_sdk_***`version`***/tools/cc32xx_tools/certificate-playground/certcatalogPlayGround20160911.lst.signed_3220.bin] を選択します。次に [開く] を選択します。

20. []
ボタンをクリックしてプロジェクトを保存します。

21. []
ボタンを選択します。

22. [Program Image (Create and Program) (プログラムイメージ (作成およびプログラム))] を選択します。

23. プログラミングプロセスが完了したら、SOP ジャンパーをピンの最初のセット (位置 = 0) に配置し、ボードをリセットし、ターミナルエミュレーターを再接続して、Code Composer Studio でデモをデバッグしたときの出力と同じであることを確認します。ターミナル出力にアプリケーションのバージョン番号を記録します。ファームウェアが OTA 更新によって更新されたことを確認するために、後でこのバージョン番号を使用します。

    ターミナルは、次のような出力を表示します。

```
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
```

Espressif ESP32 にファームウェアの初期バージョンをインストールする

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このガイドは、「Espressif ESP32-DevKitC および ESP-WROVER-KIT の開始方法」、および「無線による更新の前提条件」のステップをすでに実行していることを前提に書かれています。OTA 更新を試みる前に、ボードとツールチェーンが正しく設定されていることを確認するために FreeRTOS の開始方法で説明した MQTT デモプロジェクトを実行することができます。

初期のファクトリイメージをボードにフラッシュするには

1. *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。
2. OTA 更新の前提条件 で生成した SHA-256/ECDSA PEM 形式のコード署名証明書を vendors/*vendor*/boards/*board*/aws_demos/config_files/ota_demo_config.h にコピーします。これは、次の方法でフォーマットする必要があります。

   ```
   #define otapalconfigCODE_SIGNING_CERTIFICATE  \
   "-----BEGIN CERTIFICATE-----\n" \
   "...base64 data...\n" \
   "-----END CERTIFICATE-----\n";
   ```

3. OTA 更新デモを選択した状態で、「ESP32 の開始方法」で説明した手順と同じ手順に従って、イメージを構築してフラッシュします。以前にプロジェクトを構築してフラッシュしたことがあ

る場合、最初に make clean を実行する必要がある場合があります。make flash monitor を実行すると、次のような表示になります。デモアプリケーションは複数のタスクを一度に実行するため、メッセージの順序が異なることがあります。

```
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. ESP32 ボードは OTA 更新をリッスンしています。ESP-IDF モニタリングは、`make flash monitor` コマンドによって起動されます。終了するには、[Ctrl+] を押します。任意の TTY ターミナルプログラム (PuTTY、Tera Term、GNU Screen など) を使用して、ボードのシリアル出力をリッスンすることもできます。ボードのシリアルポートに接続すると再起動が始まる可能性があることに注意してください。

Nordic nRF52840 DK にファームウェアの初期バージョンをインストールする

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このガイドは、「Nordic nRF52840-DK の開始方法」および「無線による更新の前提条件」のステップをすでに実行していることを前提に書かれています。OTA 更新を試みる前に、ボードとツールチェーンが正しく設定されていることを確認するために FreeRTOS の開始方法で説明した MQTT デモプロジェクトを実行することができます。

初期のファクトリイメージをボードにフラッシュするには

1. *freertos*/vendors/nordic/boards/nrf52840-dk/aws_demos/config_files/aws_demo_config.h を開きます。
2. #define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED を CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED に置き換えます。
3. OTA 更新デモを選択した状態で、「Nordic nRF52840-DK の開始方法」で説明した手順と同じ手順に従って、イメージを構築してフラッシュします。

   次のような出力が表示されます。

```
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
```

ボードは OTA 更新をリッスンしています。

Windows Simulator の初期ファームウェア

Windows Simulator を使用する場合、ファームウェアの初期バージョンをフラッシュする必要はありません。Windows Simulator は aws_demos アプリケーションの一部であり、ファームウェアも含まれています。

カスタムボードにファームウェアの初期バージョンをインストールする

IDE を使用して、aws_demos プロジェクトを構築します。必ず OTA ライブラリを含めてください。FreeRTOS のソースコードの構造の詳細については、「FreeRTOS デモ」を参照してください。

FreeRTOS プロジェクトまたはお使いのデバイスに、コード署名証明書、プライベートキー、および証明書の信頼チェーンを含めてください。

適切なツールを使用して、アプリケーションをボードに焼き付け、正しく動作していることを確認します。

ファームウェアのバージョンを更新する

FreeRTOS に含まれる OTA エージェントは、更新のバージョンをチェックし、既存のファームウェアバージョンより新しい場合にのみインストールします。次の手順は、OTA デモアプリケーションのファームウェアバージョンを更新する方法を示しています。

1. IDE で aws_demos プロジェクトを開きます。
2. ファイル /vendors/*vendor*/boards/*board*/aws_demos/config_files/ota_demo_config.h を見つけて APP_VERSION_BUILD の値を増やします。
3. 0 以外のファイルタイプ (ファームウェア以外のファイル) で Renesas RX65N プラットフォームへのアップデートをスケジュールするには、Renesas Secure Flash Programmer ツールを使用してファイルに署名する必要があります。そうでない場合、デバイスの署名チェックが失敗します。このツールは、Renesas 独自のファイルタイプである拡張子 .rsu を持つ署名付きファイルパッケージを作成します。ツールは、GitHub にあります。次のコマンドの例を使用して、イメージを生成できます。

```
"Renesas Secure Flash Programmer.exe" CUI Update "RX65N(ROM 2MB)/Secure
 Bootloader=256KB" "sig-sha256-ecdsa" 1 "file_name" "output_file_name.rsu"
```

4. プロジェクトを再構築します。

ファームウェア更新を、更新を保存する Amazon S3 バケットを作成する の手順に従って作成した Amazon S3 バケットにコピーする必要があります。Amazon S3 にコピーする必要があるファイルの名前は、使用しているハードウェアプラットフォームによって異なります。

- Texas Instruments CC3220SF-LAUNCHXL: `vendors/ti/boards/cc3220_launchpad/aws_demos/ccs/debug/aws_demos.bin`
- Espressif ESP32: `vendors/espressif/boards/esp32/aws_demos/make/build/aws_demos.bin`

OTA 更新の作成 (AWS IoT コンソール)

1. AWS IoT コンソールのナビゲーションペインの「リモートアクションの管理」で、「ジョブ」を選択します。
2. [ジョブの作成] を選択します。
3. [ジョブタイプ] で [FreeRTOS OTA 更新ジョブを作成] を選択し、[次へ] を選択します。
4. [ジョブのプロパティ] で、[ジョブ名] と (オプションで) ジョブの [説明] を入力して、[次へ] を選択します。
5. OTA 更新は、1 つ以上のデバイスのグループに展開できます。[更新するデバイス] で、ドロップダウンから 1 つ以上のモノまたはモノのグループを選択します。
6. [ファームウェアイメージ転送プロトコルの選択] で、[HTTP] または [MQTT]、あるいはその両方を選択して、使用するプロトコルを各デバイスから決定できるようにします。
7. [ファイルに署名して選択する] で、[新しいファイルに署名] を選択します。
8. [コード署名プロファイル] で、[新しいプロファイルの作成] を選択します。
9. [Create a code signing profile (コード署名プロファイルの作成)] に、コード署名プロファイルの名前を入力します。

    a. [Device hardware platform (デバイスハードウェアプラットフォーム)] の下で、使用しているハードウェアプラットフォームを選択します。

> **Note**
>
> FreeRTOS に適したハードウェアプラットフォームのみがこのリストに表示されます。認定されていないプラットフォームをテストしていて、署名に ECDSA P-256

SHA-256 暗号化スイートを使用している場合は、Windows Simulator コード署名プロファイルを選択して互換性のある署名を作成できます。認定されていないプラットフォームを使用していて、署名に ECDSA P-256 SHA-256 以外の暗号スイートを使用している場合は、Code Signing for を使用するか AWS IoT、ファームウェアの更新に自分で署名できます。詳細については、「ファームウェアの更新にデジタル署名する」を参照してください。

b. [コード署名証明書] で、[既存の証明書の選択] を選択し、以前にインポートした証明書を選択するか、[新しいコード署名証明書のインポート] を選択してファイルを選択し、[インポート] を選択して新しい証明書をインポートします。

c. [Pathname of code signing certificate on device] (デバイスのコード署名証明書のパス名) に、デバイスのコード署名証明書の完全修飾パス名を入力します。ほとんどのデバイスでは、このフィールドを空欄にできます。Windows シミュレーターの場合や、証明書を特定のファイルの場所に配置するデバイスの場合は、ここにパス名を入力します。

> **Important**
>
> ファイルシステムのルートにコード署名証明書が存在する場合、Texas Instruments CC3220SF-LAUNCHXL では、ファイル名の前にスラッシュ記号 (**/**) を含めないでください。そうでない場合、OTA の更新は認証中に `file not found` エラーで失敗します。

d. [作成] を選択します。

10. [ファイル] で [既存のファイルの選択] を選択し、[S3 を参照] を選択します。Amazon S3 バケットのリストが表示されます。ファームウェア更新を含むバケットを選択し、バケット内のファームウェア更新を選択します。

> **Note**
>
> Microchip Curiosity PIC32MZEF デモプロジェクトは、デフォルト名が `mplab.production.bin` および `mplab.production.ota.bin` の、2 つのバイナリイメージを生成します。OTA 更新用の画像をアップロードする際は、2 番目のファイルを使用します。

11. [デバイス上のファイルのパス名] に、OTA ジョブがファームウェアイメージをコピーするデバイス上の場所の完全修飾パス名を入力します。この場所はプラットフォームによって異なります。

> **Important**
>
> Texas Instruments CC3220SF-LAUNCHXL では、セキュリティの制約により、ファームウェアイメージのパス名は `/sys/mcuflashimg.bin` である必要があります。

12. [ファイルタイプ] を開き、0～255 の範囲の整数値を入力します。入力したファイルタイプが、MCU に配信されるジョブドキュメントに追加されます。この値をどうするかについては、MCU ファームウェア/ソフトウェア開発者が一切の権限を持ちます。例えば、プライマリプロセッサから独立してファームウェアを更新可能なセカンダリプロセッサを持つ MCU などが考えられます。OTA 更新ジョブを受信したデバイスは、ファイルタイプを使用して、更新の対象となるプロセッサを識別できます。
13. [IAM ロール] で、「OTA 更新サービスロールを作成する」の手順に従ってロールを選択します。
14. [次へ] を選択します。
15. OTA 更新ジョブの ID と説明を入力します。
16. [Job type] (ジョブタイプ) で、[Your job will complete after deploying to the selected devices/groups (snapshot)] (ジョブは、選択したデバイス/グループへのデプロイ後に完了します (スナップショット)) を選択します。
17. ジョブに適切なオプション設定 ([ジョブ実行ロールアウト]、[ジョブ中止]、[ジョブ実行タイムアウト]、[タグ]) を選択します。
18. [作成] を選択します。

以前に署名済みのファームウェアイメージを使用するには

1. [Select and sign your firmware image (ファームウェアイメージの選択と署名)] の下で、[Select a previously signed firmware image (以前に署名済みのファームウェアイメージを選択し)] を選択します。
2. [Pathname of firmware image on device] (デバイスのファームウェアイメージのパス名) の下に、OTA ジョブがファームウェアイメージをコピーするデバイスの場所の完全修飾パス名を入力します。この場所はプラットフォームによって異なります。
3. [Previous code signing job (以前のコード署名ジョブ)] の下で、[Select (選択)] を選択し、OTA 更新に使用しているファームウェアイメージに署名するために使用した以前のコード署名ジョブを選択します。

カスタムの署名済みファームウェアイメージの使用

1. [Select and sign your firmware image (ファームウェアイメージの選択と署名)] の下で、[Use my custom signed firmware image (カスタムの署名済みファームウェアイメージの使用)] を選択します。
2. [Pathname of code signing certificate on device] (デバイスのコード署名証明書のパス名) に、デバイスのコード署名証明書の完全修飾パス名を入力します。ほとんどのデバイスでは、このフィールドを空欄にできます。Windows シミュレーターの場合や、証明書を特定のファイルの場所に配置するデバイスの場合は、ここにパス名を入力します。
3. [Pathname of firmware image on device] (デバイスのファームウェアイメージのパス名) の下に、OTA ジョブがファームウェアイメージをコピーするデバイスの場所の完全修飾パス名を入力します。この場所はプラットフォームによって異なります。
4. [Signature (署名)] で、PEM 形式の署名を貼り付けます。
5. [元のハッシュアルゴリズム] で、ファイル署名の作成時に使用されたハッシュアルゴリズムを選択します。
6. [元の暗号化アルゴリズム] で、ファイル署名の作成時に使用されたアルゴリズムを選択します。
7. [Select your firmware image in Amazon S3] (Amazon S3 のファームウェアイメージを選択) で、Amazon S3 バケット内の Amazon S3 バケットと署名付きファームウェアイメージを選択します。

コード署名情報を指定した後、OTA 更新ジョブの種類、サービスロール、および更新用の ID を指定します。

> **Note**
>
> OTA 更新のジョブ ID に個人を特定できる情報を使用しないでください。個人を特定できる情報の例は以下のとおりです。
>
> - 名前
> - IP アドレス
> - E メールアドレス
> - ロケーション
> - 銀行の情報
> - 医療情報

1. [Job type] (ジョブタイプ) で、[Your job will complete after deploying to the selected devices/groups (snapshot)] (ジョブは、選択したデバイス/グループへのデプロイ後に完了します (スナップショット)) を選択します。
2. [IAM role for OTA update job (OTA 更新ジョブの IAM ロール)] で、OTA サービスロールを選択します。
3. 英数字でジョブ ID を入力し、[Create (作成)] を選択します。

ジョブは、ステータスが IN PROGRESS の AWS IoT コンソールに表示されます。

> **Note**
>
> - AWS IoT コンソールはジョブの状態を自動的に更新しません。ブラウザの表示を更新し、状態の更新を表示してください。

シリアル UART ターミナルをデバイスに接続します。更新されたファームウェアをデバイスがダウンロードしていることを示す出力が表示されます。

デバイスは、更新されたファームウェアをダウンロードした後、再起動してからファームウェアをインストールします。UART 端末で何が起きているかを確認することができます。

コンソールを使用して OTA 更新を作成する方法を示すチュートリアルについては、「無線通信経由更新デモアプリケーション」を参照してください。

を使用した OTA 更新の作成 AWS CLI

を使用して OTA 更新 AWS CLI を作成する場合、次の操作を行います。

1. ファームウェアイメージにデジタル署名します。
2. デジタル署名されたファームウェアイメージのストリームを作成します。
3. OTA 更新ジョブを開始します。

ファームウェアの更新にデジタル署名する

を使用して OTA 更新 AWS CLI を実行する場合は、Code Signing for を使用するか AWS IoT、ファームウェア更新に自分で署名できます。Code Signing for でサポートされている暗号化署名アルゴリズムとハッシュアルゴリズムのリストについては AWS IoT、SigningConfigurationOverrides」を参照してください。Code Signing for でサポートされていない暗号化アルゴリズムを使用する場合

は AWS IoT、Amazon S3 にアップロードする前にファームウェアバイナリに署名する必要があります。

Code Signing for を使用したファームウェアイメージの署名 AWS IoT

Code Signing for を使用してファームウェアイメージに署名するには AWS IoT、 AWS SDKsまたはコマンドラインツールのいずれかを使用できます。Code Signing for の詳細については AWS IoT、「Code Signing for AWS IoT」を参照してください。

コード署名ツールをインストールして設定したら、署名されていないファームウェアイメージを Amazon S3 バケットにコピーし、次の AWS CLI コマンドを使用してコード署名ジョブを開始します。put-signing-profile コマンドは、再利用可能なコード署名プロファイルを作成します。start-signing-job コマンドは、署名ジョブを開始します。

```
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* および *your-destination-bucket-name* は、同じ Amazon S3 バケットの名前に置き換えることができます。

これらは put-signing-profile および start-signing-job コマンドのパラメータです。

`source`

S3 バケット内の符号なしファームウェアの場所を指定します。

- `bucketName`: S3 バケットの名前。

- key: S3 バケット内のファームウェアのキー (ファイル名)。
- version: S3 バケット内のファームウェアの S3 バージョン。これはファームウェアのバージョンとは異なります。ファームウェアのバージョンは、Amazon S3 コンソールを参照し、バケットを選択して、ページの上部にある [Versions] (バージョン) の横にある [Show] (表示) を選択すると表示されます。

destination

S3 バケット内の署名付きファームウェアがコピーされるデバイスの送信先。このパラメータの形式は、source パラメータと同じです。

signing-material

コード署名証明書の ARN。この ARN は、証明書を ACM にインポートするときに生成されます。

signing-parameters

署名のためのキーと値のペアのマップ。これらには、署名の際に使用する情報が含まれます。

> **Note**
>
> このパラメータは、Code Signing for AWS IoTによる OTA の更新への署名で、コード署名プロファイルを作成するときに必要になります。

platform

OTA 更新を配布するハードウェアプラットフォームの platformId。

使用可能なプラットフォームとそれらの platformId の値のリストを表示するには、aws signer list-signing-platforms コマンドを使用します。

署名ジョブが開始され、署名済みファームウェアイメージがコピー先の Amazon S3 バケットに書き込まれます。署名済みファームウェアイメージのファイル名は GUID です。ストリームを作成するときは、このファイル名が必要です。このファイル名は、Amazon S3 コンソールを参照し、バケットを選択すると表示されます。GUID ファイル名のファイルが表示されない場合は、ブラウザを更新してください。

このコマンドは、ジョブの ARN とジョブ ID を表示します。これらの値は後で必要になります。Code Signing for の詳細については AWS IoT、「Code Signing for AWS IoT」を参照してください。

手動でファームウェアイメージに署名する

ファームウェアイメージにデジタル署名し、署名済みファームウェアイメージを Amazon S3 バケットにアップロードします。

ファームウェアの更新のストリームを作成する

ストリームは、デバイスによって消費されるデータへの抽象インターフェイスです。ストリームは、異なる場所または異なるクラウドベースのサービスに保存されているデータにアクセスすることの複雑さを隠すことができます。OTA 更新マネージャーサービスを使用すると、Amazon S3 のさまざまな場所に保存されている複数のデータを使用して、OTA 更新を実行できます。

AWS IoT OTA 更新を作成するときに、署名付きファームウェア更新を含むストリームを作成することもできます。署名済みファームウェアイメージを識別する JSON ファイル (`stream.json`) を作成します。JSON ファイルには次の内容が含まれています。

```
[
  {
    "fileId":"your_file_id",
    "s3Location":{
      "bucket":"your_bucket_name",
      "key":"your_s3_object_key"
    }
  }
]
```

これらは JSON ファイル内の属性です。

`fileId`

ファームウェアイメージを識別する 0〜255 の任意の整数。

`s3Location`

ファームウェアがストリーミングするためのバケットとキー。

`bucket`

署名されていないファームウェアイメージが格納されている Amazon S3 バケット。

`key`

Amazon S3 バケット内の署名済みファームウェアイメージのファイル名。バケットの内容を調べることによって、Amazon S3 コンソールでこの値を見つけることができます。

Code Signing for を使用している場合 AWS IoT、ファイル名は Code Signing for によって生成された GUID です AWS IoT。

create-stream AWS CLI コマンドを使用して、スキーマを作成します。

```
aws iot create-stream \
    --stream-id your_stream_id \
    --description your_description \
    --files file://stream.json \
    --role-arn your_role_arn
```

コマンドの引数は次のとおりですcreate-stream AWS CLI 。

stream-id

ストリームを識別する任意の文字列。

description

ストリームの説明 (省略可能)。

files

ストリーミングするファームウェアイメージに関するデータを含む JSON ファイルへの 1 つ以上の参照。JSON ファイルには、次の属性が含まれている必要があります。

fileId

任意のファイル ID。

s3Location

署名済ファームウェアイメージが格納されているバケット名と署名済ファームウェアイメージのキー (ファイル名)。

bucket

署名済ファームウェアイメージが格納されている Amazon S3 バケット。

key

署名済ファームウェアイメージのキー (ファイル名)。

Code Signing for を使用する場合 AWS IoT、このキーは GUID です。

次は、`stream.json` ファイルの例です。

```
[
    {
        "fileId":123,
        "s3Location": {
            "bucket":"codesign-ota-bucket",
            "key":"48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
        }
    }
]
```

role-arn

ファームウェアイメージが保存されている Amazon S3 バケットへのアクセス権も付与する OTA サービスロール。

署名付きファームウェアイメージの Amazon S3 オブジェクトキーを見つけるには、 aws signer describe-signing-job --job-id ***my-job-id*** コマンドを使用します。ここで、 `my-job-id`は create-signing-job AWS CLI コマンドによって表示されるジョブ ID です。describe-signing-job コマンドの出力には、署名済ファームウェアイメージのキーが含まれています。

```
... text deleted for brevity ...
  "signedObject": {
    "s3": {
      "bucketName": "ota-bucket",
      "key": "7309da2c-9111-48ac-8ee4-5a4262af4429"
    }
  }
... text deleted for brevity ...
```

OTA 更新の作成

create-ota-update AWS CLI コマンドを使用して OTA 更新ジョブを作成します。

> **Note**
>
> OTA 更新のジョブ ID に個人を特定できる情報 (PII) を使用しないでください。個人を特定できる情報の例は以下のとおりです。
>
> - 名前
> - IP アドレス

- E メールアドレス
- ロケーション
- 銀行の情報
- 医療情報

```
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 形式

```
{
  "otaUpdateId": "string",
  "description": "string",
  "targets": [
    "string"
  ],
  "protocols": [
    "string"
  ],
  "targetSelection": "string",
  "awsJobExecutionsRolloutConfig": {
    "maximumPerMinute": "integer",
    "exponentialRate": {
      "baseRatePerMinute": "integer",
      "incrementFactor": "double",
      "rateIncreaseCriteria": {
```

```
        "numberOfNotifiedThings": "integer",
        "numberOfSucceededThings": "integer"
      }
    }
  },
  "awsJobPresignedUrlConfig": {
    "expiresInSec": "long"
  },
  "awsJobAbortConfig": {
    "abortCriteriaList": [
      {
        "failureType": "string",
        "action": "string",
        "thresholdPercentage": "double",
        "minNumberOfExecutedThings": "integer"
      }
    ]
  },
  "awsJobTimeoutConfig": {
    "inProgressTimeoutInMinutes": "long"
  },
  "files": [
    {
      "fileName": "string",
      "fileType": "integer",
      "fileVersion": "string",
      "fileLocation": {
        "stream": {
          "streamId": "string",
          "fileId": "integer"
        },
        "s3Location": {
          "bucket": "string",
          "key": "string",
          "version": "string"
        }
      },
      "codeSigning": {
        "awsSignerJobId": "string",
        "startSigningJobParameter": {
          "signingProfileParameter": {
            "certificateArn": "string",
            "platform": "string",
            "certificatePathOnDevice": "string"
```

```
          },
          "signingProfileName": "string",
          "destination": {
            "s3Destination": {
              "bucket": "string",
              "prefix": "string"
            }
          }
        },
        "customCodeSigning": {
          "signature": {
            "inlineDocument": "blob"
          },
          "certificateChain": {
            "certificateName": "string",
            "inlineDocument": "string"
          },
          "hashAlgorithm": "string",
          "signatureAlgorithm": "string"
        }
      },
      "attributes": {
        "string": "string"
      }
    }
  ],
  "roleArn": "string",
  "additionalParameters": {
    "string": "string"
  },
  "tags": [
    {
      "Key": "string",
      "Value": "string"
    }
  ]
}
```

cli-input-json フィールド

名前	型	説明
otaUpdateId	string	作成する OTA 更新の ID。

名前	型	説明
	(最大: 128 最小: 1)	
description	文字列 (最大: 2028)	OTA 更新の説明。
targets	リスト	OTA 更新を受信するターゲットデバイス。
protocols	リスト	OTA 更新イメージの転送に使用されるプロトコル。有効な値は、[HTTP]、[MQTT]、[HTTP, MQTT] です。HTTP と MQTT の両方が指定されている場合、対象のデバイスはプロトコルを選ぶことができます。

名前	型	説明
targetSelection	文字列	更新が継続して実行される (CONTINUOUS) か、ターゲットとして指定されたすべてのモノが更新を完了した後に完了する (SNAPSHOT) かどうかを指定します。CONTINUOUS の場合は、ターゲットで変更が検出されたときに、モノで更新を実行することもできます。たとえば、グループ内に元からあったすべてのモノによって更新が完了した後でも、そのモノがターゲットグループに追加されると、そのモノで更新が実行されます。有効な値: CONTINUOUS | SNAPSHOT enum: CONTINUOUS | SNAPSHOT
awsJobExecutionsRolloutConfig		OTA 更新のロールアウトのための設定。
maximumPerMinute	integer (最大: 1000 最小: 1)	1 分ごとに開始される OTA 更新ジョブの実行の最大数。
exponentialRate		ジョブのロールアウトに対する増加レート。このパラメータによりジョブのロールアウトに対する指数レートの上昇を定義できます。

名前	型	説明
baseRatePerMinute	integer (最大: 1000 最小: 1)	保留中のジョブを 1 分間で通知する最小数 (ジョブのロールアウト開始時)。これは、ロールアウトの初期レートです。
rateIncreaseCriteria		ジョブのロールアウトでレートの増加を開始する基準。 AWS IoT は、10 進数の 1 桁後までサポートします (たとえば、1.5 はサポートしますが、1.55 はサポートしません)。
numberOfNotifiedThings	integer (最小: 1)	この数のモノが通知されると、ロールアウトレートの上昇が開始します。
numberOfSucceededThings	integer (最小: 1)	この数のモノがジョブの実行時に成功すると、ロールアウトレートの上昇が開始します。
awsJobPresignedUrlConfig		署名付き URL に関する構成情報。
expiresInSec	long	署名付き URL が有効である時間の長さ (秒) です。有効な値は 60 から 3600 で、規定値は 1800 秒です。署名付き URL は、ジョブドキュメントのリクエストを受信したときに生成されます。

名前	型	説明
awsJobAbortConfig		ジョブの停止がいつどのように行われるかを決定する基準。
abortCriteriaList	リスト	ジョブをいつどのように停止するかを決定する基準のリスト。
failureType	文字列	ジョブの停止を開始できるジョブ実行失敗のタイプ。 enum: FAILED | REJECTED | TIMED_OUT | ALL
action	文字列	ジョブの停止を開始するために実行するジョブアクションのタイプ。 enum: CANCEL
minNumberOfExecutedThings	integer (最小: 1)	ジョブを停止する前に、ジョブ実行通知を受け取る必要があるモノの最小数。
awsJobTimeoutConfig		各デバイスがジョブの実行を終了する必要のある時間を指定します。ジョブの実行ステータスが IN_PROGRESS に設定されると、タイマーが開始されます。タイマーが時間切れになるまでにジョブの実行ステータスが別の終了ステータスに設定されない場合は、自動的に TIMED_OUT に設定されます。

名前	型	説明
inProgressTimeoutInMinutes	long	このデバイスがこのジョブの実行を終了する必要のある時間を分単位で指定します。タイムアウト間隔は 1 分〜 7 日 (1 〜 10,080 分) の範囲で指定できます。進捗タイマーは更新できず、ジョブのすべての実行に適用されます。ジョブの実行がこの間隔より長時間、IN_PROGRESS ステータスのままになるたびに、ジョブの実行は失敗し、終了ステータス TIMED_OUT に切り替わります。
files	リスト	OTA 更新によってストリーミングされるファイル。
fileName	文字列	ファイルの名前。
fileType	integer range- max:255 min:0	クラウドから受信したファイルの種類をデバイスで識別するために、ジョブドキュメントに含めることができる整数値。
fileVersion	文字列	ファイルのバージョン。
fileLocation		更新されたファームウェアの場所。
stream		OTA 更新を含むストリーム。
streamId	文字列 (最大: 128 最小: 1)	ストリーム ID。

名前	型	説明
fileId	integer (最大: 255 最小: 0)	ストリームに関連付けられたファイルの ID。
s3Location		S3 での更新されたファームウェアの場所。
bucket	文字列 (最小: 1)	S3 バケット。
key	文字列 (最小: 1)	S3 キー。
version	文字列	S3 バケットのバージョン。
codeSigning		ファイルのコード署名方法。
awsSignerJobId	文字列	ファイルに署名するために作成された AWSSignerJob の ID。
startSigningJobParameter		コード署名ジョブの説明。
signingProfileParameter		コード署名プロファイルの説明。
certificateArn	文字列	証明書の ARN。
platform	文字列	デバイスのハードウェアプラットフォーム。
certificatePathOnDevice	文字列	デバイス上のコード署名証明書の場所。
signingProfileName	文字列	コード署名プロファイルの名前。

名前	型	説明
destination		コード署名付きファイルを書き込む場所。
s3Destination		S3 での更新されたファームウェアの場所の説明。
bucket	文字列 (最小: 1)	更新されたファームウェアを含む S3 バケット。
prefix	文字列	S3 プレフィックス。
customCodeSigning		ファイルにコードを署名するためのカスタムメソッド。
signature		ファイルの署名。
inlineDocument	blob	コード署名の base64 でエンコードされたバイナリ表現。
certificateChain		証明書チェーン。
certificateName	文字列	証明書の名前
inlineDocument	文字列	署名証明書チェーンの base64 でエンコードされたバイナリ表現。
hashAlgorithm	文字列	ファイルに符号を付けるために使用されるハッシュアルゴリズム。
signatureAlgorithm	文字列	ファイルに符号を付けるために使用される署名アルゴリズム。
attributes	マップ	名前と属性のペアのリスト。

名前	型	説明
roleArn	文字列 (最大: 2048 最小: 20)	OTA 更新ジョブを作成するための Amazon S3、 AWS IoT ジョブ、および AWS Code Signing リソース AWS IoT へのアクセスを許可する IAM ロール。
additionalParameters	マップ	名前と値のペアである追加の OTA 更新パラメータのリスト。
tags	リスト	アップデートを管理するために使用できるメタデータ。
Key	文字列 (最大: 128 最小: 1)	タグのキー。
Value	文字列 (最大: 256 最小: 1)	タグの値。

Output

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

AWS CLI 出力フィールド

名前	型	説明
otaUpdateId	string	OTA 更新 ID。

名前	型	説明
	(最大: 128 最小: 1)	
awsIotJobId	文字列	OTA 更新に関連付けられた AWS IoT ジョブ ID。
otaUpdateArn	文字列	OTA 更新 ARN。
awsIotJobArn	文字列	OTA 更新に関連付けられた AWS IoT ジョブ ARN。
otaUpdateStatus	文字列	OTA 更新のステータス。 enum: CREATE_PENDING | CREATE_IN_PROGRESS | CREATE_COMPLETE | CREATE_FAILED

以下は、Code Signing を使用する create-ota-update コマンドに渡される JSON ファイルの例です AWS IoT。

```
[
  {
    "fileName": "firmware.bin",
    "fileType": 1,
    "fileLocation": {
      "stream": {
        "streamId": "004",
        "fileId":123
      }
    },
    "codeSigning": {
      "awsSignerJobId": "48c67f3c-63bb-4f92-a98a-4ee0fbc2bef6"
    }
  }
]
```

以下は、インラインファイルを使用してカスタムコード署名マテリアルを提供する create-ota-update AWS CLI コマンドに渡される JSON ファイルの例です。

```
[
  {
    "fileName": "firmware.bin",
    "fileType": 1,
    "fileLocation": {
      "stream": {
        "streamId": "004",
        "fileId": 123
      }
    },
    "codeSigning": {
      "customCodeSigning":{
        "signature":{
          "inlineDocument":"your_signature"
        },
        "certificateChain": {
          "certificateName": "your_certificate_name",
          "inlineDocument":"your_certificate_chain"
        },
        "hashAlgorithm":"your_hash_algorithm",
        "signatureAlgorithm":"your_signature_algorithm"
      }
    }
  }
]
```

以下は、FreeRTOS OTA がコード署名ジョブを開始し、コード署名プロファイルとストリームを作成できるようにする create-ota-update AWS CLI コマンドに渡される JSON ファイルの例です。

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "your_bucket_name",
        "key": "your_object_key",
        "version": "your_S3_object_version"
      }
    },
    "codeSigning":{
      "startSigningJobParameter":{
```

```
        "signingProfileName": "myTestProfile",
        "signingProfileParameter": {
          "certificateArn": "your_certificate_arn",
          "platform": "your_platform_id",
          "certificatePathOnDevice": "certificate_path"
        },
        "destination": {
          "s3Destination": {
            "bucket": "your_destination_bucket"
          }
        }
      }
    }
  }
]
```

以下は、既存のプロファイルでコード署名ジョブを開始し、指定されたストリームを使用する OTA 更新を作成する create-ota-update AWS CLI コマンドに渡される JSON ファイルの例です。

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "your_s3_bucket_name",
        "key": "your_object_key",
        "version": "your_S3_object_version"
      }
    },
    "codeSigning":{
      "startSigningJobParameter":{
        "signingProfileName": "your_unique_profile_name",
        "destination": {
          "s3Destination": {
            "bucket": "your_destination_bucket"
          }
        }
      }
    }
  }
]
```

以下は、FreeRTOS OTA が既存のコード署名ジョブ ID でストリームを作成できるようにする create-ota-update AWS CLI コマンドに渡される JSON ファイルの例です。

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

以下は、OTA 更新を作成する create-ota-update AWS CLI コマンドに渡される JSON ファイルの例です。この更新では、指定された S3 オブジェクトからストリームが作成され、カスタムコード署名が使用されます。

```
[
  {
    "fileName": "your_firmware_path_on_device",
    "fileType": 1,
    "fileVersion": "1",
    "fileLocation": {
      "s3Location": {
        "bucket": "your_bucket_name",
        "key": "your_object_key",
        "version": "your_S3_object_version"
      }
    },
    "codeSigning":{
      "customCodeSigning": {
        "signature":{
          "inlineDocument":"your_signature"
        },
        "certificateChain": {
          "inlineDocument":"your_certificate_chain",
          "certificateName": "your_certificate_path_on_device"
        },
        "hashAlgorithm":"your_hash_algorithm",
        "signatureAlgorithm":"your_sig_algorithm"
      }
    }
```

```
  }
]
```

OTA 更新を一覧表示する

list-ota-updates AWS CLI コマンドを使用して、すべての OTA 更新のリストを取得できます。

```
aws iot list-ota-updates
```

コマンド list-ota-updates の出力は次のようになります。

```
{
  "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
    }
  ]
}
```

OTA 更新に関する情報の取得

get-ota-update AWS CLI コマンドを使用して、OTA 更新の作成または削除ステータスを取得できます。

```
aws iot get-ota-update --ota-update-id your-ota-update-id
```

get-ota-update コマンドからの出力は以下のようになります。

```
{
```

```
    "otaUpdateInfo": {
        "otaUpdateId": "ota-update-001",
        "otaUpdateArn": "arn:aws:iot:region:123456789012:otaupdate/ota-update-001",
        "creationDate": 1575414146.286,
        "lastModifiedDate": 1575414149.091,
        "targets": [
            "arn:aws:iot:region:123456789012:thing/myDevice"
        ],
        "protocols": [ "HTTP" ],
        "awsJobExecutionsRolloutConfig": {
            "maximumPerMinute": 0
        },
        "awsJobPresignedUrlConfig": {
            "expiresInSec": 1800
        },
        "targetSelection": "SNAPSHOT",
        "otaUpdateFiles": [
            {
                "fileName": "my_firmware.bin",
                "fileType": 1,
                "fileLocation": {
                    "s3Location": {
                        "bucket": "my-bucket",
                        "key": "my_firmware.bin",
                        "version": "AvP3bfJC9gyqnwoxPHuTqM5GWENt4iii"
                    }
                },
                "codeSigning": {
                    "awsSignerJobId": "b7a55a54-fae5-4d3a-b589-97ed103737c2",
                    "startSigningJobParameter": {
                        "signingProfileParameter": {},
                        "signingProfileName": "my-profile-name",
                        "destination": {
                            "s3Destination": {
                                "bucket": "some-ota-bucket",
                                "prefix": "SignedImages/"
                            }
                        }
                    },
                    "customCodeSigning": {}
                }
            }
        ],
        "otaUpdateStatus": "CREATE_COMPLETE",
```

```
        "awsIotJobId": "AFR_OTA-ota-update-001",
        "awsIotJobArn": "arn:aws:iot:region:123456789012:job/AFR_OTA-ota-update-001"
    }
}
```

otaUpdateStatus に戻される値には次のものがあります。

CREATE_PENDING

OTA 更新の作成は保留中です。

CREATE_IN_PROGRESS

OTA 更新が作成されています。

CREATE_COMPLETE

OTA 更新が作成されました。

CREATE_FAILED

OTA 更新の作成に失敗しました。

DELETE_IN_PROGRESS

OTA 更新が削除中です。

DELETE_FAILED

OTA 更新の削除に失敗しました。

> **Note**
>
> OTA 更新の作成後にその実行ステータスを取得するには、describe-job-execution コマンドを使用する必要があります。詳細については、ジョブ実行の説明を参照してください。

OTA 関連データの削除

現在、 AWS IoT コンソールを使用してストリームや OTA 更新を削除することはできません。を使用して AWS CLI 、ストリーム、OTA 更新、および OTA 更新中に作成された AWS IoT ジョブを削除できます。

OTA ストリームを削除する

MQTT を使用する OTA 更新を作成する場合、コマンドラインまたは AWS IoT コンソールを使用してストリームを作成し、ファームウェアをチャンクに分割して MQTT 経由で送信できます。次の例に示すように、 delete-stream AWS CLI コマンドを使用してこのストリームを削除できます。

```
aws iot delete-stream --stream-id your_stream_id
```

OTA ストリームの削除

OTA 更新の作成時、以下が作成されます。

- OTA 更新ジョブデータベース内のエントリ。
- 更新を実行する AWS IoT ジョブ。
- 更新する各デバイスの AWS IoT ジョブ実行。

delete-ota-update コマンドは、OTA 更新ジョブデータベース内のエントリのみを削除します。 AWS IoT ジョブを削除するには、delete-job コマンドを使用する必要があります。

delete-ota-update コマンドを使用して、OTA 更新を削除します。

```
aws iot delete-ota-update --ota-update-id your_ota_update_id
```

ota-update-id

削除する OTA 更新の ID。

delete-stream

OTA 更新に関連付けられたストリームを削除します。

force-delete-aws-job

OTA 更新に関連付けられた AWS IoT ジョブを削除します。このフラグが設定されておらず、ジョブが `In_Progress` 状態の場合、ジョブは削除されません。

OTA 更新用に作成された IoT ジョブを削除する

FreeRTOS は、OTA 更新を作成するときに AWS IoT ジョブを作成します。ジョブの実行は、ジョブを処理するデバイスごとにも作成されます。delete-job AWS CLI コマンドを使用して、ジョブとそれに関連するジョブ実行を削除できます。

```
aws iot delete-job --job-id your-job-id --no-force
```

no-force パラメータは、終了状態のジョブ (COMPLETED または CANCELLED) のみを削除できるように指定します。force パラメータを渡すことによって、非ターミナルステータスにあるジョブを削除することができます。詳細については、DeleteJob API を参照してください。

> **Note**
>
> ステータスが IN_PROGRESS のジョブを削除すると、デバイスの IN_PROGRESS であるジョブの実行が中断され、デバイスが非決定的な状態になる場合があります。削除されたジョブを実行している各デバイスが既知の状態に回復できることを確認します。

ジョブ用に作成されたジョブ実行の数およびその他の要素に応じて、ジョブの削除に時間がかかる場合があります。ジョブが削除されている間、そのステータスは DELETION_IN_PROGRESS です。ステータスがすでに DELETION_IN_PROGRESS のジョブを削除あるいはキャンセルしようとすると、エラーになります。

delete-job-execution を使用してジョブの実行を削除できます。いくらかのデバイスがジョブを処理できない場合、ジョブの実行を削除することをお勧めします。これにより、次の例に示すように、単一のデバイスのジョブ実行が削除されます。

```
aws iot delete-job-execution --job-id your-job-id --thing-name
                    your-thing-name --execution-number your-job-execution-number --no-
force
```

delete-job AWS CLI コマンドと同様に、 --forceパラメータを に渡delete-job-executionして、ジョブ実行を強制的に削除できます。詳細については、DeleteJobExecution API を参照してください。

> **Note**
>
> ステータスが IN_PROGRESS のジョブの実行を削除すると、デバイスの IN_PROGRESS であるジョブの実行が中断され、デバイスが非決定的な状態になる場合があります。削除されたジョブを実行している各デバイスが既知の状態に回復できることを確認します。

OTA 更新デモアプリケーションを使用する方法の詳細については、「無線通信経由更新デモアプリケーション」を参照してください。

OTA 更新マネージャーサービス

無線 (OTA) による更新マネージャーサービスを使用すると、次のことができます。

- OTA 更新と、 AWS IoT ジョブ、 AWS IoT ストリーム、コード署名など、使用するリソースを作成します。
- OTA 更新に関する情報を取得します。
- AWS アカウントに関連付けられているすべての OTA 更新を一覧表示します。
- OTA 更新を削除します。

OTA 更新は、OTA 更新マネージャサービスによって保持されるデータ構造です。以下を含みます。

- OTA 更新 ID。
- オプションの OTA 更新の説明。
- 更新するデバイスのリスト (targets)。
- OTA 更新のタイプ: CONTINUOUS または SNAPSHOT。必要な更新の種類についての説明は、AWS IoT 開発者ガイドの「ジョブ」セクションを参照してください。
- OTA 更新を実行するために使用されるプロトコル: [MQTT]、[HTTP]、または [MQTT, HTTP]。MQTT と HTTP を指定すると、デバイス設定によって使用されるプロトコルが決定されます。
- ターゲットデバイスに送信するファイルのリスト。
- OTA 更新ジョブを作成するための Amazon S3、 AWS IoT ジョブ、および AWS Code Signing リソース AWS IoT へのアクセスを許可する IAM ロール。
- ユーザー定義の名前と値のペアのオプションリスト。

OTA 更新はデバイスファームウェアを更新するように設計されていますが、それらを使用して、1 つ以上のデバイスを登録する任意のファイルを送信できます AWS IoT。ファームウェアの更新を無線で送信する場合は、更新を受信するデバイスが途中で改ざんされていないことを確認できるように、デジタル署名することをお勧めします。

選択した設定に応じて、HTTP または MQTT のプロトコルを使用して、更新されたファームウェアイメージを送信できます。Code Signing for FreeRTOS を使用してファームウェアの更新に署名することも、独自のコード署名ツールを使用することもできます。

プロセスをより詳細に制御するには、MQTT 経由で更新を送信するときに CreateStream API を使用してストリームを作成できます。場合によっては、FreeRTOS エージェントコードを変更して、送受信するブロックのサイズを調整できます。

OTA 更新を作成すると、OTA マネージャーサービスでは、更新が利用可能であることをデバイスに通知する AWS IoT ジョブを作成します。FreeRTOS OTA エージェントはデバイスで動作し、更新メッセージをリッスンします。更新が利用可能になると、HTTP または MQTT 経由でファームウェア更新イメージをリクエストし、ファイルをローカルに保存します。ダウンロードしたファイルのデジタル署名をチェックし、有効な場合は、ファームウェアの更新をインストールします。FreeRTOS を使用していない場合は、独自の OTA エージェントを実装し、更新をリッスンしてダウンロードし、インストールオペレーションを実行する必要があります。

アプリケーションへの OTA エージェントの統合

無線通信 (OTA) エージェントは、OTA 更新機能を製品に追加するために記述する必要があるコードの量を簡素化するように設計されています。この統合の負担は、主に OTA エージェントの初期化と、OTA エージェントイベントメッセージに応答するためのカスタムコールバック関数の作成です。OS の初期化中に、MQTT、HTTP (HTTP がファイルのダウンロードに使用されている場合)、およびプラットフォーム固有の実装 (PAL) インターフェイスが OTA エージェントに渡されます。バッファを初期化して OTA エージェントに渡すこともできます。

> **Note**
>
> アプリケーションに OTA 更新機能を統合するのは簡単ですが、OTA の更新システムでは、単にデバイスコードの統合以上のことを理解する必要があります。モノ、認証情報、コード署名証明書、プロビジョニングデバイス、OTA 更新ジョブを使用して AWS AWS IoT アカウントを設定する方法を理解するには、FreeRTOS の前提条件」を参照してください。

接続管理

OTA エージェントは、 AWS IoT サービスに関連するすべてのコントロール通信オペレーションに MQTT プロトコルを使用しますが、MQTT 接続は管理しません。OTA エージェントがアプリケーションの接続管理ポリシーを妨げないようにするには、切断と再接続機能を含む MQTT 接続をメインのユーザーアプリケーションで処理する必要があります。このファイルは MQTT または HTTP プロトコルでダウンロードできます。OTA ジョブを作成するときに、プロトコルを選択できます。MQTT を選択した場合、OTA エージェントは制御オペレーションとファイルのダウンロードに同じ接続を使用します。

シンプルな OTA デモ

以下は、エージェントが MQTT ブローカーに接続して OTA エージェントを初期化する方法を示すシンプルな OTA デモの抜粋です。この例ではデモ用に、デフォルトの OTA アプリケーションコールバックを使用し、1 秒ごとに統計情報をいくつか返す設定をします。簡潔にするために、このデモから一部の詳細を省略します。

OTA デモでは、切断コールバックをモニタリングし、接続を再確立する MQTT 接続管理も実演します。このデモでは、切断発生後に、まず OTA エージェントのオペレーションを中断し、次に MQTT 接続の再確立を試みます。MQTT 再接続試行までの時間間隔は、最大値まで指数関数的に増えていき、ジッターも追加されます。接続が再確立されると、OTA エージェントによるオペレーションが続行されます。

AWS IoT MQTT ブローカーを使用する実例については、 demos/ota ディレクトリの OTA デモコードを参照してください。

OTA エージェントは独自のタスクなので、この例の意図的な 1 秒の遅れはこのアプリケーションにのみ影響します。エージェントのパフォーマンスに影響はありません。

```
static BaseType_t prvRunOTADemo( void )
{
    /* Status indicating a successful demo or not. */
    BaseType_t xStatus = pdFAIL;

    /* OTA library return status. */
    OtaErr_t xOtaError = OtaErrUninitialized;

    /* OTA event message used for sending event to OTA Agent.*/
    OtaEventMsg_t xEventMsg = { 0 };

    /* OTA interface context required for library interface functions.*/
    OtaInterfaces_t xOtaInterfaces;

    /* OTA library packet statistics per job.*/
    OtaAgentStatistics_t xOtaStatistics = { 0 };

    /* OTA Agent state returned from calling OTA_GetState.*/
    OtaState_t xOtaState = OtaAgentStateStopped;

    /* Set OTA Library interfaces.*/
    prvSetOtaInterfaces( &xOtaInterfaces );
```

```
    /************************** Init OTA Library. **************************/

    if( ( xOtaError = OTA_Init( &xOtaBuffer,
                                &xOtaInterfaces,
                                ( const uint8_t * ) ( democonfigCLIENT_IDENTIFIER ),
                                prvOtaAppCallback ) ) != OtaErrNone )
    {
        LogError( ( "Failed to initialize OTA Agent, exiting = %u.",
                    xOtaError ) );
    }
    else
    {
        xStatus = pdPASS;
    }

    /************************ Create OTA Agent Task. ************************/

    if( xStatus == pdPASS )
    {
        xStatus = xTaskCreate( prvOTAAgentTask,
                               "OTA Agent Task",
                               otaexampleAGENT_TASK_STACK_SIZE,
                               NULL,
                               otaexampleAGENT_TASK_PRIORITY,
                               NULL );

        if( xStatus != pdPASS )
        {
            LogError( ( "Failed to create OTA agent task:" ) );
        }
    }

    /***************************** Start OTA *****************************/

    if( xStatus == pdPASS )
    {
        /* Send start event to OTA Agent.*/
        xEventMsg.eventId = OtaAgentEventStart;
        OTA_SignalEvent( &xEventMsg );
    }

    /******************** Loop and display OTA statistics ********************/

    if( xStatus == pdPASS )
```

```
    {
        while( ( xOtaState = OTA_GetState() ) != OtaAgentStateStopped )
        {
            /* Get OTA statistics for currently executing job. */
            if( xOtaState != OtaAgentStateSuspended )
            {
                OTA_GetStatistics( &xOtaStatistics );

                LogInfo( ( " Received: %u   Queued: %u   Processed: %u   Dropped: %u",
                           xOtaStatistics.otaPacketsReceived,
                           xOtaStatistics.otaPacketsQueued,
                           xOtaStatistics.otaPacketsProcessed,
                           xOtaStatistics.otaPacketsDropped ) );
            }

            vTaskDelay( pdMS_TO_TICKS( otaexampleEXAMPLE_TASK_DELAY_MS ) );
        }
    }

    return xStatus;
}
```

このデモアプリケーションのハイレベルな流れは次のとおりです。

- MQTT エージェントコンテキストを作成します。
- AWS IoT エンドポイントに接続します。
- OTA エージェントを初期化します。
- OTA 更新ジョブを許可し、1 秒に 1 回の統計を出力するループ。
- MQTT が切断されると、OTA エージェントのオペレーションを中断します。
- 指数関数的に増える時間間隔とジッターを使用して再度接続を試みます。
- 再接続されると、OTA エージェントのオペレーションを再開します。
- エージェントが停止した場合は、1 秒遅らせて再接続を試みます。

OTA エージェントイベントにアプリケーションコールバックを使用する

前の例では、prvOtaAppCallback を OTA エージェントイベントのコールバックハンドラーとして使用しました。(OTA_Init API コールの 4 番目のパラメータを参照)。完了イベントのカスタム処理を実装する場合は、OTA デモ/アプリケーションでデフォルトの処理を変更する必要があります。OTA プロセス中、OTA エージェントは次のイベント列挙値の 1 つをコールバックハンドラーに

送信できます。これらのイベントをどのように処理するかは、アプリケーション開発者が決定します。

```
/**
 * @ingroup ota_enum_types
 * @brief OTA Job callback events.
 *
 * After an OTA update image is received and authenticated, the agent calls the user
 * callback (set with the @ref OTA_Init API) with the value OtaJobEventActivate to
 * signal that the device must be rebooted to activate the new image. When the device
 * boots, if the OTA job status is in self test mode, the agent calls the user callback
 * with the value OtaJobEventStartTest, signaling that any additional self tests
 * should be performed.
 *
 * If the OTA receive fails for any reason, the agent calls the user callback with
 * the value OtaJobEventFail instead to allow the user to log the failure and take
 * any action deemed appropriate by the user code.
 *
 * See the OtaImageState_t type for more information.
 */
typedef enum OtaJobEvent
{
    OtaJobEventActivate = 0,       /*!< @brief OTA receive is authenticated and ready
 to activate. */
    OtaJobEventFail = 1,           /*!< @brief OTA receive failed. Unable to use this
 update. */
    OtaJobEventStartTest = 2,      /*!< @brief OTA job is now in self test, perform
 user tests. */
    OtaJobEventProcessed = 3,      /*!< @brief OTA event queued by OTA_SignalEvent is
 processed. */
    OtaJobEventSelfTestFailed = 4, /*!< @brief OTA self-test failed for current job. */
    OtaJobEventParseCustomJob = 5, /*!< @brief OTA event for parsing custom job
 document. */
    OtaJobEventReceivedJob = 6,    /*!< @brief OTA event when a new valid AFT-OTA job
 is received. */
    OtaJobEventUpdateComplete = 7, /*!< @brief OTA event when the update is completed.
 */
    OtaLastJobEvent = OtaJobEventStartTest
} OtaJobEvent_t;
```

OTA エージェントは、メインアプリケーションのアクティブな処理中にバックグラウンドで更新を受け取ることができます。これらのイベントを配信する目的は、アクションを即時に実行できるかどうか、または他のアプリケーション固有の処理が完了するまで遅延する必要があるかどうかをアプリ

ケーションが判断できるようにすることです。これによって、ファームウェアの更新後のリセットなどによるアクティブな処理中 (バキューム処理など) に、予期しないデバイスの中断を防ぐことができます。これらは、コールバックハンドラーによって受信されたたジョブイベントです。

OtaJobEventActivate

このイベントがコールバックハンドラーによって受信された場合、すぐにデバイスをリセットするか、後でデバイスをリセットするようコールをスケジュールすることができます。これにより、必要に応じてデバイスのリセットとセルフテストフェーズを延期できます。

OtaJobEventFail

このイベントがコールバックハンドラーによって受信されると、更新は失敗します。この場合、何もする必要はありません。ログメッセージを出力するか、アプリケーション固有の処理を行うことができます。

OtaJobEventStartTest

セルフテストフェーズは、新しく更新されたファームウェアが正常に機能しているかどうかを判断する前に、実行およびテストを行い、更新されたファームウェアを最新の永続的なアプリケーションイメージにコミットするように意図されています。新しい更新が受信および認証され、デバイスがリセットされると、テストの準備ができ次第、OTA エージェントは OtaJobEventStartTest イベントをコールバック関数に送信します。開発者は、更新後にデバイスファームウェアが正しく機能しているかどうかを判断するために必要なテストを追加できます。デバイスファームウェアが自己テストによって信頼できると見なされた場合、コードは OTA_SetImageState(OtaImageStateAccepted) 関数を呼び出すことで、ファームウェアを新しい永続イメージとしてコミットする必要があります。

OtaJobEventProcessed

OTA_SignalEvent によってキューに入れられた OTA イベントが処理されるため、OTA バッファの解放などのクリーンアップオペレーションを実行できます。

OtaJobEventSelfTestFailed

現在のジョブで OTA セルフテストが失敗しました。このイベントのデフォルトの処理では、OTA エージェントをシャットダウンして再起動し、デバイスが前のイメージにロールバックされます。

OtaJobEventUpdateComplete

OTA ジョブ更新完了の通知イベント。

OTA セキュリティ

無線通信 (OTA) セキュリティの 3 つの側面を次に示します。

接続の安全性

OTA 更新マネージャーサービスは、 AWS IoTで使用される Transport Layer Security (TLS) 相互認証などの既存のセキュリティメカニズムに依存しています。OTA 更新トラフィックは AWS IoT デバイスゲートウェイを通過し、セキュリティメカニズムを使用します AWS IoT 。デバイスゲートウェイを介した各送受信 HTTP または MQTT メッセージは、厳密な認証と認可を受けます。

OTA 更新の真正性と完全性

ファームウェアは、信頼できるソースからのものであり、改ざんされていないことを保証するために、OTA 更新前にデジタル署名が可能です。

FreeRTOS OTA Update Manager サービスは、Code Signing for AWS IoT を使用してファームウェアに自動的に署名します。詳細については、「Code Signing for AWS IoT」を参照してください。

OTA エージェントは、デバイス上で実行され、ファームウェアがデバイスに到着したときにファームウェアの整合性チェックを実行します。

運用者のセキュリティ

コントロールプレーン API を介して行われるすべての API 呼び出しは、標準の IAM 署名バージョン 4 の認証と認可を受けます。デプロイを作成するには、`CreateDeployment`、`CreateJob`、および `CreateStream` API を呼び出す権限が必要です。さらに、Amazon S3 バケットポリシーまたは ACL では、Amazon S3 に保存されているファームウェア更新にストリーミング中にアクセスできるように、 AWS IoT サービスプリンシパルに読み取りアクセス許可を付与する必要があります。

のコード署名 AWS IoT

AWS IoT コンソールは Code Signing for AWS IoT を使用して、 でサポートされているデバイスのファームウェアイメージに自動的に署名します AWS IoT。

Code Signing for は、ACM にインポートする証明書とプライベートキー AWS IoT を使用します。テストには自己署名証明書が使えますが、信頼された商用証明機関 (CA) から証明書を取得することをお勧めします。

コード署名証明書は X.509 バージョン 3 の `Key Usage` および `Extended Key Usage` 拡張機能を使用します。`Key Usage` 拡張機能は `Digital Signature` に設定され、`Extended Key Usage` 拡張機能は `Code Signing` に設定されます。コードイメージの署名の詳細については、「Code Signing for AWS IoT Developer Guide」および「Code Signing for AWS IoT API Reference」を参照してください。

> **Note**
>
> Tools for Amazon Web Services から Code Signing for AWS IoT SDK をダウンロードできます。

OTA のトラブルシューティング

以下のセクションでは、OTA 更新で発生する問題のトラブルシューティングに関する情報が含まれています。

トピック

- OTA 更新のための CloudWatch Logs のセットアップ
- による OTA API AWS IoT コールのログ記録 AWS CloudTrail
- を使用して CreateOTAUpdate の失敗の詳細を取得する AWS CLI
- を使用して OTA 障害コードを取得する AWS CLI
- 複数のデバイスの OTA 更新のトラブルシューティング
- Texas Instruments CC3220SF Launchpad を使用した OTA 更新のトラブルシューティング

OTA 更新のための CloudWatch Logs のセットアップ

OTA 更新サービスは、Amazon CloudWatch でのログ記録をサポートしています。 AWS IoT コンソールを使用して、OTA 更新の Amazon CloudWatch ログ記録を有効にして設定できます。詳細については、Cloudwatch Logs を参照してください。

ログ記録を有効にするには、IAM ロールを作成し、OTA 更新ログを設定する必要があります。

> **Note**
>
> OTA 更新ログ記録を有効にする前に、CloudWatch Logs へのアクセス権限を理解しておく必要があります。CloudWatch Logs に対するアクセス権限のあるユーザーは、デバッグ情報を

表示できます。詳細については、「Amazon CloudWatch Logs の認証とアクセスコントロール」を参照してください。

ログ記録ロールの作成およびログ記録の有効化

AWS IoT コンソールを使用して、ログ記録ロールを作成し、ログ記録を有効にします。

1. ナビゲーションパネルから [Settings (設定)] を選択します。
2. [Logs (ログ)] の下で、[Edit (編集)] を選択します。
3. [Level of verbosity (詳細レベル)] の下で、[Debug (デバッグ)] を選択します。
4. [Set role] (ロールの設定) で、[Create new] (新規作成) を選択し、ログ記録用に IAM ロールを作成します。
5. [Name (名前)] の下にロールの一意の名前を入力します。必要なすべての権限でロールが作成されます。
6. [更新] を選択します。

OTA 更新ログ

OTA 更新サービスは、以下のいずれかが発生するとアカウントにログを生成します。

- OTA 更新が作成される。
- OTA 更新が完了する。
- コード署名ジョブが作成される。
- コード署名ジョブが完了する。
- AWS IoT ジョブが作成されます。
- AWS IoT ジョブが完了しました。
- ストリームが作成される。

CloudWatch コンソールでログを表示できます。

CloudWatch Logs で OTA 更新を表示するには

1. ナビゲーションペインで、[Logs (ログ)] を選択します。
2. [ロググループ] で、[AWSIoTLogsV2] を選択します。

OTA 更新ログには、次のプロパティが含まれます。

accountId

ログが生成された AWS アカウント ID。

actionType

ログを生成したアクションです。これは、次のいずれかの値に設定できます。

- `CreateOTAUpdate`: OTA 更新が作成されました。
- `DeleteOTAUpdate`: OTA 更新が削除されました。
- `StartCodeSigning`: コード署名ジョブが開始されました。
- `CreateAWSJob`: AWS IoT ジョブが作成されました。
- `CreateStream`: ストリームが作成されました。
- `GetStream`: ストリームのリクエストが AWS IoT MQTT ベースのファイル配信機能に送信されました。
- `DescribeStream`: ストリームに関する情報のリクエストが、 AWS IoT MQTT ベースのファイル配信機能に送信されました。

awsJobId

ログを生成した AWS IoT ジョブ ID。

clientId

ログを生成するリクエストを行った MQTT クライアント ID。

clientToken

ログを生成するリクエストに関連付けられたクライアントのトークン。

details

ログを生成したオペレーションに関する詳細。

logLevel

ログのログ記録レベル。OTA 更新ログの場合、これは常に `DEBUG` に設定されます。

otaUpdateId

ログを生成した OTA 更新の ID。

protocol

ログを生成するリクエストを行うために使用されたプロトコル。

ステータス

ログを生成したオペレーションのステータス。次の値を指定できます:

- Success (成功)
- 失敗

streamId

ログを生成した AWS IoT ストリーム ID。

timestamp

ログが生成された時刻。

topicName

ログを生成するリクエストを行うために使用された MQTT トピック。

ログの例

コード署名ジョブの開始時に生成されるログの例を次に示します。

```
{
    "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."
}
```

AWS IoT ジョブの作成時に生成されるログの例を次に示します。

```
{
    "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."
```

```
}
```

OTA 更新の作成時に生成されるログの例を次に示します。

```
{
    "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."
}
```

ストリームの作成時に生成されるログの例を次に示します。

```
{
    "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."
}
```

OTA 更新の削除時に生成されるログの例を次に示します。

```
{
    "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."
}
```

デバイスが MQTT ベースのファイル配信機能からストリームをリクエストしたときに生成されるログの例を次に示します。

```
{
    "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."
}
```

デバイスが DescribeStream API を呼び出すときに生成される口グの例を次に示します。

```
{
    "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."
}
```

による OTA API AWS IoT コールのログ記録 AWS CloudTrail

FreeRTOS は、OTA API AWS IoT コールをキャプチャし、指定した Amazon S3 バケットにログファイルを配信するサービスである CloudTrail と統合されています。CloudTrail は、コードから OTA API への API AWS IoT コールをキャプチャします。 APIs CloudTrail で収集された情報を使用して、OTA AWS IoT に対するリクエスト、リクエスト元のソース IP アドレス、リクエスト者、リクエスト日時などを判断できます。

CloudTrail を設定して有効にする方法などの詳細については、AWS CloudTrail ユーザーガイドを参照してください。https://docs.aws.amazon.com/awscloudtrail/latest/userguide/

CloudTrail での FreeRTOS 情報

AWS アカウントで CloudTrail ログ記録が有効になっている場合、OTA AWS IoT アクションに対する API コールは CloudTrail ログファイルで追跡され、他の AWS サービスレコードとともに書き込まれます。CloudTrail は、期間とファイルサイズに基づいて、新しいファイルをいつ作成して書き込むかを決定します。

次の AWS IoT OTA コントロールプレーンアクションは CloudTrail によってログに記録されます。

- CreateStream
- DescribeStream
- ListStreams
- UpdateStream
- DeleteStream
- CreateOTAUpdate
- GetOTAUpdate
- ListOTAUpdates
- DeleteOTAUpdate

> **Note**
>
> AWS IoT OTA データプレーンアクション (デバイス側) は CloudTrail によってログに記録されません。これらをモニタリングするために CloudWatch を使用します。

各ログエントリには、リクエストの生成者に関する情報が含まれます。ログエントリのユーザーアイデンティティ情報は、次のことを確認するのに役立ちます。

- リクエストがルートまたは IAM ユーザー認証情報で行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーの一時的なセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって行われたかどうか。

詳細については、「 AWS IoT OTA API リファレンス」のCloudTrail userIdentity Element. AWS IoT OTA アクション」を参照してください。

必要な場合はログファイルを自身の Amazon S3 バケットに保存できますが、ログファイルを自動的にアーカイブまたは削除するように Amazon S3 ライフサイクルルールを定義することもできます。デフォルトでは Amazon S3 のサーバー側の暗号化 (SSE) を使用して、ログファイルが暗号化されます。

ログファイルの配信時に通知を受け取る場合は、Amazon SNS 通知が発行されるように CloudTrail を設定できます。詳細については、CloudTrail 用の Amazon SNS 通知の構成を参照してください。

複数の AWS リージョンと複数の AWS アカウントからの AWS IoT OTA ログファイルを 1 つの Amazon S3 バケットに集約することもできます。

詳細は、CloudTrail ログファイルを複数のリージョンから受け取ると CloudTrail ログファイルを複数のアカウントから受け取るを参照してください。

FreeRTOS ログファイルのエントリについて

CloudTrail ログファイルには、1 つ以上のログエントリを含むことができます。各エントリには、複数の JSON 形式のイベントがリストされます。ログエントリは任意の送信元からの単一のリクエストを表し、リクエストされたアクション、アクションの日時、リクエストのパラメータなどに関する情報が含まれます。ログエントリは、パブリック API コールの順序付けられたスタックトレースではないため、特定の順序では表示されません。

以下の例は、`CreateOTAUpdate` アクションへの呼び出しからのログを示す CloudTrail ログエントリを示しています。

```
{
    "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"
}
```

を使用して CreateOTAUpdate の失敗の詳細を取得する AWS CLI

OTA 更新ジョブの作成プロセスが失敗した場合に、問題を解決するために実行可能なアクションがあることがあります。OTA 更新ジョブを作成すると、OTA マネージャーサービスは IoT ジョブを作成し、ターゲットデバイス用にスケジュールします。このプロセスでは、アカウント内の他のタイプ

の AWS リソース (コード署名ジョブ、 AWS IoT ストリーム、Amazon S3 オブジェクト) も作成または使用します。エラーが発生すると、 AWS IoT ジョブを作成せずにプロセスが失敗する可能性があります。このトラブルシューティングセクションでは、障害の詳細を取得する方法について説明します。

1. AWS CLI をインストールして設定します。
2. `aws configure` を実行し、以下の情報を入力します。

```
$ aws configure
AWS Access Key ID [None]: AccessID
AWS Secret Access Key [None]: AccessKey
Default region name [None]: Region
Default output format [None]: json
```

詳細については、aws configure を使用したクイック設定を参照してください。

3. 以下を実行します:

```
aws iot get-ota-update --ota-update-id ota_update_job_001
```

ota_update_job_001 は OTA 更新の作成時に指定した ID です。

4. 出力は次のようになります。

```
{
    "otaUpdateInfo": {
        "otaUpdateId": "ota_update_job_001",
        "otaUpdateArn":
 "arn:aws:iot:region:account_id:otaupdate/ota_update_job_001",
        "creationDate": 1584646864.534,
        "lastModifiedDate": 1584646865.913,
        "targets": [
            "arn:aws:iot:region:account_id:thing/thing_001"
        ],
        "protocols": [
            "MQTT"
        ],
        "awsJobExecutionsRolloutConfig": {},
        "awsJobPresignedUrlConfig": {},
        "targetSelection": "SNAPSHOT",
        "otaUpdateFiles": [
            {
```

```
                "fileName": "/12ds",
                 "fileLocation": {
                     "s3Location": {
                         "bucket": "bucket_name",
                         "key": "demo.bin",
                         "version": "Z7X.TWSAS7JSi4rybc02nMdcE41W1tV3"
                     }
                 },
                 "codeSigning": {
                     "startSigningJobParameter": {
                         "signingProfileParameter": {},
                         "signingProfileName": "signing_profile_name",
                         "destination": {
                             "s3Destination": {
                                 "bucket": "bucket_name",
                                 "prefix": "SignedImages/"
                             }
                         }
                     },
                     "customCodeSigning": {}
                 }
             }
         ],
         "otaUpdateStatus": "CREATE_FAILED",
         "errorInfo": {
             "code": "AccessDeniedException",
             "message": "S3 object demo.bin not accessible. Please check
 your permissions (Service: AWSSigner; Status Code: 403; Error Code:
 AccessDeniedException; Request ID: 01d8e7a1-8c7c-4d85-9fd7-dcde975fdd2d)"
         }
     }
}
```

作成に失敗した場合、コマンド出力の otaUpdateStatus フィールドに CREATE_FAILED が含まれ、errorInfo フィールドに障害の詳細が含まれます。

を使用して OTA 障害コードを取得する AWS CLI

1. AWS CLIをインストールして設定します。
2. aws configure を実行し、以下の情報を入力します。

```
$ aws configure
```

```
AWS Access Key ID [None]: AccessID
AWS Secret Access Key [None]: AccessKey
Default region name [None]: Region
Default output format [None]: json
```

詳細については、aws configure を使用したクイック設定を参照してください。

3. 以下を実行します:

```
aws iot describe-job-execution --job-id JobID --thing-name ThingName
```

JobID はステータスを取得するジョブの完全なジョブ ID 文字列で (作成時に OTA 更新ジョブに関連付けられていました) 、*ThingName* はデバイスが に登録されている AWS IoT モノの名前です。 AWS IoT

4. 出力は次のようになります。

```
{
    "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
    }
}
```

この出力例では、「detailsmap」の「reason」には 2 つのフィールドがあります。「0xEEEEEEEE」として表示されるフィールドには、OTA Agent からの一般的なエラーコードが含まれています。「0xffffffff」として表示されるフィールドには、サブコードが含まれています。一般的なエラーコードの一覧は、https://docs.aws.amazon.com/freertos/latest/lib-ref/html1/aws__ota__agent_8h.html にあります。プレフィックス「kOTA_Err_」が付いたエラーコード

を参照してください。サブコードは、プラットフォーム固有のコードであるか、一般的なエラーに関する詳細を提供します。

複数のデバイスの OTA 更新のトラブルシューティング

同じファームウェアイメージを使用して複数のデバイス (モノ) で OTA を実行するには、不揮発性メモリから `clientcredentialIOT_THING_NAME` を取得する関数 (たとえば `getThingName()`) を実装します。この関数が、OTA によって上書きされない不揮発性メモリの一部からモノ名を読み取ること、および最初のジョブを実行する前にそのモノ名がプロビジョニングされていることを確認してください。JITP フローを使用している場合は、デバイス証明書の共通名からモノ名を読み取ることができます。

Texas Instruments CC3220SF Launchpad を使用した OTA 更新のトラブルシューティング

CC3220SF Launchpad プラットフォームは、ソフトウェア改ざん検出メカニズムを提供します。また、整合性違反があると増分されるセキュリティ警告カウンターを使用します。セキュリティアラートカウンターが事前定義されたしきい値 (デフォルトは 15) に達し、ホストが非同期イベント `SL_ERROR_DEVICE_LOCKED_SECURITY_ALERT` を受信すると、デバイスはロックされます。その場合、ロックされたデバイスはアクセスが制限されます。デバイスを回復するには、デバイスを再プログラムするか、または工場出荷時に復元プロセスを使用してファクトリイメージに戻すことができます。`network_if.c` 内の非同期イベントハンドラを更新し、目的の動作をプログラムします。

FreeRTOS ライブラリ

FreeRTOS ライブラリは、FreeRTOS カーネルとその内部ライブラリに機能を追加します。組み込みアプリケーションでのネットワーキングとセキュリティのために FreeRTOS ライブラリを使用できます。FreeRTOS ライブラリを使用すると、アプリケーションは AWS IoT サービスとやり取りすることもできます。FreeRTOS には、以下のことが可能なライブラリが含まれています。

- MQTT とデバイスシャドウを使用して、デバイスを AWS IoT クラウドに安全に接続します。
- AWS IoT Greengrass コアを検出して接続します。
- Wi-Fi 接続を管理します。
- FreeRTOS 無線通信経由更新をリッスンして処理します。

`libraries` ディレクトリには、FreeRTOS ライブラリのソースコードがあります。ライブラリ機能の実装を支援するヘルパー関数があります。これらのヘルパー関数を変更することはお勧めしません。

FreeRTOS 移植ライブラリ

以下の移植ライブラリは、FreeRTOS コンソールでダウンロード可能な FreeRTOS の設定に含まれています。これらのライブラリはプラットフォームに依存します。それらの内容は、ハードウェアプラットフォームに応じて変わります。これらのライブラリをデバイスに移植する方法については、FreeRTOS 移植ガイドを参照してください。

FreeRTOS 移植ライブラリ

ライブラリ	API リファレンス	説明
Bluetooth Low Energy	Bluetooth Low Energy API リファレンス	FreeRTOS Bluetooth Low Energy ライブラリを使用すると、マイクロコントローラーはゲートウェイデバイスを介して AWS IoT MQTT ブローカーと通信できます。詳細については、「Bluetooth Low Energy ライブラリ」を参照してください。
無線による更新	AWS IoT Over-the-air更新 API リファレンス	FreeRTOS AWS IoT Over-the-air (OTA) 更新ライブラリを使用すると、FreeRTOS デバイスで更新通知の管理、更新のダウンロード、ファームウェア更新の暗号化検証を実行できます。 詳細については、「AWS IoT 無線通信経由 (OTA) ライブラリ」を参照してください。
FreeRTOS+POSIX	FreeRTOS+POSIX API リファレンス	FreeRTOS+POSIX ライブラリを使用して、POSIX 準拠のアプリケーションを FreeRTOS エコシステムに移植できます。 詳細については、「FreeRTOS+POSIX」を参照してください。

ライブラリ	API リファレンス	説明
セキュアソケット	セキュアソケット API リファレンス	詳細については、「セキュアソケットライブラリ」を参照してください。
FreeRTOS+TCP	FreeRTOS+TCP API リファレンス	FreeRTOS+TCP は、FreeRTOS のスケーラブルなオープンソースでスレッドセーフな TCP/IP スタックです。 詳細については、「FreeRTOS+TCP」を参照してください。
Wi-Fi	Wi-Fi API リファレンス	FreeRTOS Wi-Fi ライブラリを使用すると、マイクロコントローラーの低いレベルのワイヤレスのスタックとの連動が可能になります。 詳細については、「Wi-Fi ライブラリ」を参照してください。
corePKCS11		corePKCS11 ライブラリは、プロビジョニングと TLS クライアント認証をサポートするための公開鍵暗号標準 #11 のリファレンス実装です。 詳細については、「corePKCS11 ライブラリ」を参照してください。
TLS		詳細については、「Transport Layer Security」を参照してください。
共通 I/O	共通 I/O API リファレンス	詳細については、「共通 I/O」を参照してください。

ライブラリ	API リファレンス	説明
セルラーインターフェイス	セルラーインターフェイス API リファレンス	セルラーインターフェイスライブラリは、統一された API を使用して、いくつかの一般的なセルラーモデムの機能を公開します。詳細については、「セルラーインターフェイスライブラリ」を参照してください。

FreeRTOS アプリケーションライブラリ

オプションで、以下のスタンドアロンアプリケーションライブラリを FreeRTOS 設定に含めて、クラウド上の AWS IoT サービスとやり取りできます。

> **Note**
>
> 一部のアプリケーションライブラリには、 AWS IoT Device SDK for Embedded C のライブラリと同じ APIs があります。 これらのライブラリについては、AWS IoT 「 Device SDK C API Reference」を参照してください。 AWS IoT Device SDK for Embedded C の詳細については、「」を参照してくださいAWS IoT 埋め込み C 用の Device SDK。

FreeRTOS アプリケーションライブラリ

ライブラリ	API リファレンス	説明
AWS IoT Device Defender	Device Defender C SDK API リファレンス	FreeRTOS AWS IoT Device Defender ライブラリは FreeRTOS デバイスを に接続します AWS IoT Device Defender。 詳細については、「AWS IoT Device Defender ライブラリ」を参照してください。
AWS IoT Greengrass	Greengrass API リファレンス	FreeRTOS AWS IoT Greengrass ライブラリは FreeRTOS デバイスを

ライブラリ	API リファレンス	説明
		に接続します AWS IoT Greengrass。 詳細については、「AWS IoT Greengrass 検出ライブラリ」を参照してください。
MQTT	MQTT (v1.x.x) ライブラリ API リファレンス MQTT (v1) Agent API リファレンス MQTT (v2.x.x) C SDK API リファレンス	coreMQTT ライブラリは、FreeRTOS デバイスが MQTT トピックを公開し、サブスクライブするためのクライアントを提供します。MQTT は、デバイスが操作するために使用するプロトコルです AWS IoT。 coreMQTT ライブラリバージョン 3.0.0 の詳細については、「coreMQTT ライブラリ」を参照してください。
coreMQTT エージェント	coreMQTT エージェントライブラリ API リファレンス	coreMQTT エージェントライブラリは、スレッドセーフを coreMQTT ライブラリに追加する高レベル API です。これにより、バックグラウンドで MQTT 接続を管理し、他のタスクからの介入を必要としない専用の MQTT エージェントタスクを作成できます。このライブラリでは coreMQTT の API と同等のスレッドセーフを提供するため、マルチスレッド環境で使用できます。 coreMQTT エージェントライブラリの詳細については、coreMQTT エージェントライブラリを参照してください。

ライブラリ	API リファレンス	説明
AWS IoT デバイスシャドウ	Device Shadow C SDK API リファレンス	AWS IoT Device Shadow ライブラリを使用すると、FreeRTOS デバイスが AWS IoT デバイスシャドウとやり取りできるようになります。 詳細については、「AWS IoT Device Shadow ライブラリ」を参照してください。

FreeRTOS ライブラリの設定

FreeRTOS および AWS IoT Device SDK for Embedded C の構成設定は、C プリプロセッサ定数として定義されます。グローバル設定ファイルで、または`gcc` の `-D` などのコンパイラオプションを使用して、構成を設定することができます。構成設定はコンパイル時定数として定義されているため、構成設定が変更された場合はライブラリを再ビルドする必要があります。

グローバル設定ファイルを使用して、設定オプションを設定する場合は、`iot_config.h` という名前でファイルを作成して保存し、インクルードパスに配置します。ファイル内で、`#define` ディレクティブを使用して、FreeRTOS ライブラリ、デモ、テストを設定します。

サポートされているグローバル設定オプションの詳細については、「グローバル設定ファイルリファレンス」を参照してください。

backoffAlgorithm ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

backoffAlgorithm ライブラリは、同じデータブロックを繰り返し再送信するためのスペースを確保して、ネットワークの混雑を回避するためのユーティリティライブラリです。このライブラリは、ジッターを伴うエクスポネンシャルバックオフアルゴリズムを使ってネットワークオペレーション (サーバーとのネットワーク接続の失敗など) を再試行するためのバックオフ時間を算出します。

通常、ジッターを伴うエクスポネンシャルバックオフは、ネットワークの混雑またはサーバーの高負荷によって引き起こされるサーバー接続の失敗やネットワーク要求を再試行するときに使用されます。ネットワーク接続を同時に試行する複数のデバイスによって作成される再試行要求のタイミングを分散する目的で使用されます。接続の状態が悪い環境では、クライアントが切断される可能性が常につきまといます。そのため、バックオフ戦略は、接続できる可能性が低いときに繰り返し再接続を試みないことで、バッテリーを節約するのに役立ちます。

ライブラリは C 言語で記述されており、ISO C90 と MISRA C:2012 に準拠するように設計されています。ライブラリは標準 C ライブラリ以外のライブラリに依存せず、ヒープ割り当てもありません。そのため、IoT マイクロコントローラに適しているだけでなく、他のプラットフォームに完全に移植することもできます。

このライブラリは無償で使用でき、MIT オープンソースライセンスに基づいて配布されます。

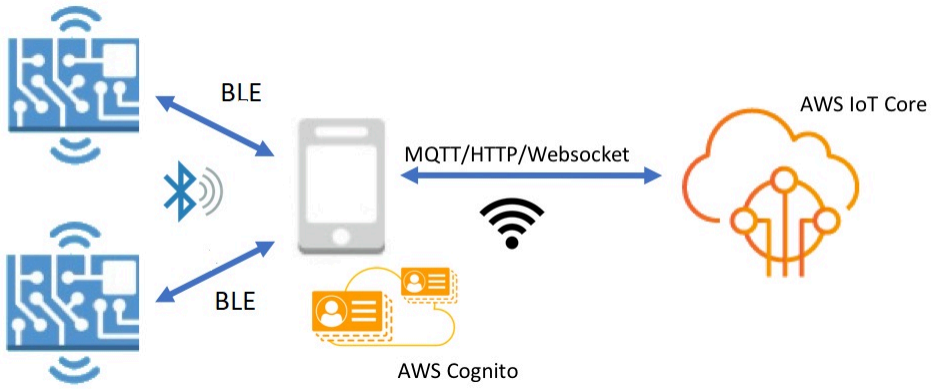
backoffAlgorithm のコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
backoff_algorithm.c	0.1 K	0.1 K
合計 (概算)	0.1 K	0.1 K

Bluetooth Low Energy ライブラリ

> **Important**
>
> このライブラリは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

概要

FreeRTOS は、携帯電話などのプロキシデバイスを使用した Bluetooth Low Energy 経由での Message Queuing Telemetry Transport (MQTT) トピックのパブリッシュおよびサブスクライブをサポートします。FreeRTOS Bluetooth Low Energy (BLE) ライブラリを使用すると、マイクロコントローラーは AWS IoT MQTT ブローカーと安全に通信できます。

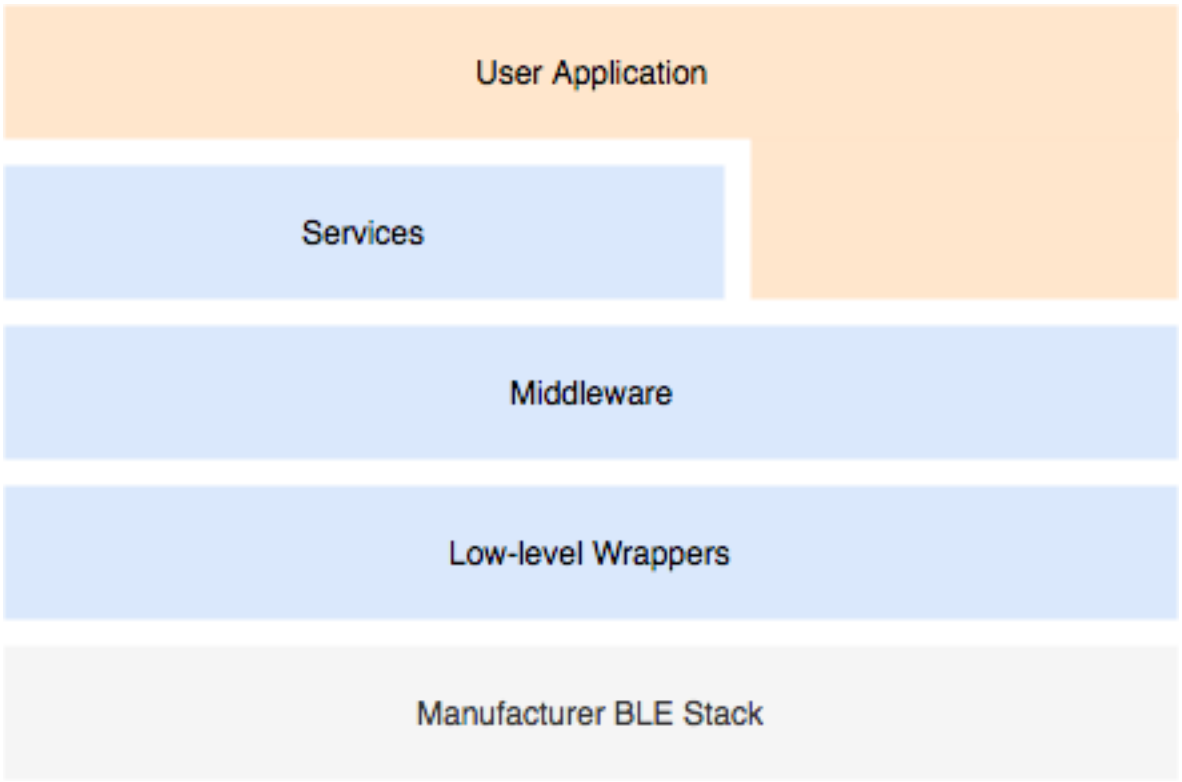


FreeRTOS Bluetooth デバイス用 Mobile SDK を使用すると、マイクロコントローラー上の組み込みアプリケーションと BLE 経由で通信するネイティブモバイルアプリケーションを作成できます。この Mobile SDK の詳細については、「FreeRTOS Bluetooth デバイス用の Mobile SDK」を参照してください。

FreeRTOS BLE ライブラリには、Wi-Fi ネットワークの設定、大量のデータの転送、および BLE 経由でのネットワーク抽象化の提供を行うためのサービスが含まれています。FreeRTOS BLE ライブラリには、BLE スタックをより直接的に制御するためのミドルウェアと低レベルの API も含まれています。

アーキテクチャ

FreeRTOS BLE ライブラリは、サービス、ミドルウェア、および低レベルラッパーの 3 つのレイヤーで構成されます。



サービス

FreeRTOS BLE サービスレイヤーは、ミドルウェア API を活用する 4 つの汎用属性 (GATT) サービスで構成されています。

- デバイス情報
- Wi-Fi プロビジョニング
- ネットワークの抽象化
- 大容量オブジェクトの転送

デバイス情報

デバイス情報サービスは、以下を含む、マイクロコントローラーに関する詳細情報を収集します。

- デバイスが使用する FreeRTOS のバージョン。
- デバイスが登録されているアカウントの AWS IoT エンドポイント。
- Bluetooth Low Energy の最大送信単位 (MTU)。

Wi-Fi プロビジョニング

Wi-Fi プロビジョニングサービスでは、Wi-Fi 機能を備えたマイクロコントローラーで次のことができます。

- 範囲内のネットワークを一覧表示します。
- ネットワークとネットワーク認証情報をフラッシュメモリに保存します。
- ネットワークの優先度を設定します。
- フラッシュメモリからネットワークとネットワーク認証情報を削除します。

ネットワークの抽象化

ネットワーク抽象化サービスは、アプリケーションのネットワーク接続タイプを抽象化します。一般的な API は、デバイスの Wi-Fi、イーサネット、および Bluetooth Low Energy ハードウェアスタックとやり取りして、アプリケーションが複数の接続タイプと互換性を持てるようにします。

大容量オブジェクトの転送

大容量オブジェクトの転送サービスは、クライアントとの間でデータを送受信します。Wi-Fi プロビジョニングやネットワークの抽象化などの他のサービスでは、大容量オブジェクトの転送サービスを使用してデータを送受信します。大容量オブジェクトの転送 API を使用して、サービスを直接操作することもできます。

MQTT over BLE

MQTT over BLE には、BLE 経由で MQTT プロキシサービスを作成するための GATT プロファイルが含まれています。MQTT プロキシサービスを使用すると、MQTT クライアントはゲートウェイデバイスを介して AWS MQTT ブローカーと通信できます。例えば、 プロキシサービスを使用して、FreeRTOS を実行しているデバイスをスマートフォンアプリ経由で AWS MQTT に接続できます。BLE デバイスは GATT サーバーであり、ゲートウェイデバイス向けのサービスと特性を公開します。GATT サーバーは、公開されたこれらのサービスと特性を使用して、そのデバイスのクラウドとの MQTT 操作を実行します。詳細については、「付録 A: MQTT over BLE の GATT プロファイル」を参照してください。

ミドルウェア

FreeRTOS Bluetooth Low Energy ミドルウェアは、低レベルの API からの抽象化です。ミドルウェア API は、Bluetooth Low Energy スタックにユーザーが使いやすいインターフェイスを構成します。

ミドルウェア API を使用すると、複数のレイヤーにわたる複数のコールバックを 1 つのイベントに登録できます。Bluetooth Low Energy ミドルウェアを初期化すると、サービスも初期化され、広告を開始します。

柔軟なコールバックサブスクリプション

Bluetooth Low Energy ハードウェアが切断され、MQTT over Bluetooth Low Energy サービスが切断を検出する必要があるとします。作成したアプリケーションも、同じ切断イベントを検出する必要があります。Bluetooth Low Energy ミドルウェアは、上位レイヤーが低レベルのリソースと競合することなく、コールバックを登録したコードの異なる部分にイベントをルーティングできます。

低レベルのラッパー

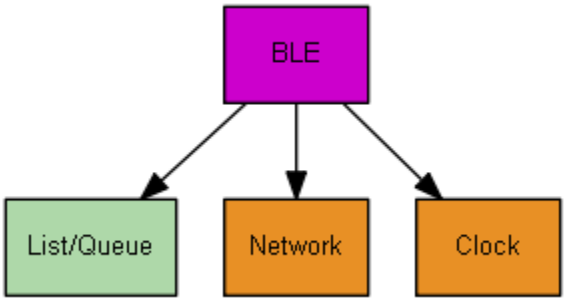
低レベルの FreeRTOS Bluetooth Low Energy ラッパーは、製造元の Bluetooth Low Energy スタックからの抽象化です。低レベルのラッパーは、ハードウェアを直接制御するための共通の API セットを提供します。低レベルの API は RAM の使用を最適化しますが、機能は限られています。

Bluetooth Low Energy サービス API を使用して、Bluetooth Low Energy サービスを操作します。サービス API は、低レベル API よりも多くのリソースを要求します。

依存関係と要件

Bluetooth Low Energy ライブラリには次の直接的な依存関係があります。

- 線形コンテナライブラリ
- スレッド管理、タイマー、クロック関数、およびネットワークアクセスのためにオペレーティングシステムと連結するプラットフォームレイヤー。



Wi-Fi プロビジョニングサービスにのみ、次のような FreeRTOS ライブラリの依存関係があります。

GATT のサービス	依存関係
Wi-Fi プロビジョニング	Wi-Fi ライブラリ

AWS IoT MQTT ブローカーと通信するには、 AWS アカウントがあり、デバイスを AWS IoT モノとして登録する必要があります。セットアップの詳細については、「AWS IoT 開発者ガイド」を参照してください。

FreeRTOS Bluetooth Low Energy は、モバイルデバイスでのユーザー認証に Amazon Cognito を使用します。MQTT プロキシサービスを使用するには、Amazon Cognito アイデンティティとユーザープールを作成する必要があります。各 Amazon Cognito アイデンティティには、適切なポリシーがアタッチされている必要があります。詳細については、「Amazon Cognito デベロッパーガイド」をご覧ください。

ライブラリ設定ファイル

FreeRTOS MQTT over Bluetooth Low Energy サービスを使用するアプリケーションは、設定パラメータが定義されている `iot_ble_config.h` ヘッダーファイルを提供する必要があります。定義されていない設定パラメータは、`iot_ble_config_defaults.h` で指定されたデフォルト値を使用します。

次のような重要な設定パラメータがあります。

`IOT_BLE_ADD_CUSTOM_SERVICES`

ユーザーに独自のサービスの作成を許可します。

`IOT_BLE_SET_CUSTOM_ADVERTISEMENT_MSG`

ユーザーに広告およびスキャン応答メッセージのカスタマイズを許可します。

詳細については、「Bluetooth Low Energy API リファレンス」を参照してください。

最適化

ボードのパフォーマンスを最適化する場合は、次の点を考慮してください。

- 低レベルの API はあまり RAM を使用しませんが、機能は限られています。
- `iot_ble_config.h` ヘッダーファイルの `bleconfigMAX_NETWORK` パラメータをより低い値に設定すると、消費されるスタックの量を減らすことができます。

- MTU サイズを最大値まで拡大してメッセージのバッファリングを制限し、コードの実行速度を向上させ、RAM の消費を抑えることができます。

使用制限

デフォルトでは、FreeRTOS Bluetooth Low Energy ライブラリは eBTpropertySecureConnectionOnly プロパティを TRUE に設定し、デバイスをセキュア接続のみモードにします。Bluetooth コア仕様 v5.0、ボリューム 3、パート C、10.2.4 で指定されているように、デバイスがセキュア接続のみモードにある場合、最も低い LE セキュリティモード 1 レベル、レベル 1 より高いアクセス許可を持つ属性へのアクセスには、最も高い LE セキュリティモード 1 レベル、レベル 4 が必要です。LE セキュリティモード 1 レベル 4 では、デバイスは数値比較のための入出力機能を備えている必要があります。

ここでは、サポートされているモード、およびその関連プロパティを示します。

モード 1、レベル 1 (セキュリティなし)

```
/* 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 )
```

モード 1、レベル 2 (暗号化のある認証されていないペアリング)

```
#define IOT_BLE_ENABLE_NUMERIC_COMPARISON        ( 0 )
#define IOT_BLE_ENABLE_SECURE_CONNECTION         ( 0 )
#define IOT_BLE_INPUT_OUTPUT                     ( eBTIONone )
```

モード 1、レベル 3 (暗号化のある認証されたペアリング)

このモードはサポートされていません。

モード 1、レベル 4 (暗号化のある認証された LE のセキュアな接続ペアリング)

このモードはデフォルトでサポートされています。

LE セキュリティモードの詳細については、「Bluetooth コア仕様 v5.0、ボリューム 3、パート C、10.2.1」を参照してください。

初期化

アプリケーションがミドルウェアを介して Bluetooth Low Energy スタックを操作する場合は、ミドルウェアを初期化するだけで済みます。ミドルウェアは、スタックの下位レイヤーの初期化を行います。

ミドルウェア

ミドルウェアを初期化するには

1. Bluetooth Low Energy ミドルウェア API を呼び出す前に、Bluetooth Low Energy ハードウェアドライバーを初期化します。
2. Bluetooth Low Energy を有効にします。
3. `IotBLE_Init()` でミドルウェアを初期化します。

> **Note**
>
> AWS デモを実行する場合、この初期化ステップは必要ありません。デモの初期化は、*freertos*`/demos/network_manager` にあるネットワークマネージャによって処理されます。

低レベル API

FreeRTOS Bluetooth Low Energy GATT サービスを使用しない場合は、ミドルウェアをバイパスし、低レベル API を直接操作してリソースを節約できます。

低レベル API を初期化するには

1. API を呼び出す前に、Bluetooth Low Energy ハードウェアドライバーを初期化します。ドライバーの初期化は、Bluetooth Low Energy 低レベル API の一部ではありません。
2. Bluetooth Low Energy 低レベル API は、能力とリソースを最適化するために Bluetooth Low Energy スタックへの呼び出しを有効/無効にします。API を呼び出す前に Bluetooth Low Energy を有効にする必要があります。

```
const BTInterface_t * pxIface = BTGetBluetoothInterface();
xStatus = pxIface->pxEnable( 0 );
```

3. Bluetooth マネージャーには、Bluetooth Low Energy と Bluetooth classic の両方に共通の API が含まれています。共通マネージャーのコールバックは、2 番目に初期化する必要があります。

```
xStatus = xBTInterface.pxBTInterface->pxBtManagerInit( &xBTManagerCb );
```

4. Bluetooth Low Energy アダプタは、共通 API の上部に適合します。共通 API を初期化したように、コールバックを初期化する必要があります。

```
xBTInterface.pxBTLeAdapterInterface = ( BTBleAdapter_t * )
 xBTInterface.pxBTInterface->pxGetLeAdapter();
xStatus = xBTInterface.pxBTLeAdapterInterface-
>pxBleAdapterInit( &xBTBleAdapterCb );
```

5. 新しいユーザーアプリケーションの登録

```
xBTInterface.pxBTLeAdapterInterface->pxRegisterBleApp( pxAppUuid );
```

6. GATT サーバーへのコールバックを初期化します。

```
xBTInterface.pxGattServerInterface = ( BTGattServerInterface_t * )
 xBTInterface.pxBTLeAdapterInterface->ppvGetGattServerInterface();
xBTInterface.pxGattServerInterface->pxGattServerInit( &xBTGattServerCb );
```

Bluetooth Low Energy アダプタを初期化すると、GATT サーバーを追加できます。GATT サーバーは一度に 1 つしか登録できません。

```
xStatus = xBTInterface.pxGattServerInterface->pxRegisterServer( pxAppUuid );
```

7. セキュアな接続のみ、MTU サイズなどのアプリケーションプロパティを設定します。

```
xStatus = xBTInterface.pxBTInterface-
>pxSetDeviceProperty( &pxProperty[ usIndex ] );
```

API リファレンス

完全な API リファレンスについては、「Bluetooth Low Energy API リファレンス」を参照してください。

使用例

以下の例では、新規サービスの広告と作成のために Bluetooth Low Energy ライブラリを使用する方法を示します。FreeRTOS Bluetooth Low Energy デモアプリケーションの詳細については、Bluetooth Low Energy デモアプリケーションを参照してください。

広告

1. アプリケーションで、広告 UUID を設定します。

```
static const BTUuid_t _advUUID =
{
    .uu.uu128 = IOT_BLE_ADVERTISING_UUID,
    .ucType   = eBTuuidType128
};
```

2. 次に、IotBle_SetCustomAdvCb コールバック関数を定義します。

```
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;
}
```

   このコールバックにより、広告メッセージの UUID とスキャン応答のフルネームを送信します。

3. vendors/*vendor*/boards/*board*/aws_demos/config_files/iot_ble_config.h を開き、IOT_BLE_SET_CUSTOM_ADVERTISEMENT_MSG を 1 に設定します。これにより IotBle_SetCustomAdvCb コールバックがトリガーされます。

新しいサービスの追加

サービスの詳細な例については、*freertos*/.../ble/services を参照してください。

1. サービスの特性と記述子の UUID を作成します。

```
#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. 特性と記述子のハンドルを登録するバッファを作成します。

```
static uint16_t usHandlesBuffer[egattDemoNbAttributes];
```

3. 属性テーブルを作成します。いくつかの RAM を保存するには、テーブルを const として定義します。

> **Important**
>
> 属性は常に、サービスを最初の属性として、順序どおりに作成します。

```
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. コールバックの配列を作成します。このコールバックの配列は、上で定義されたテーブル配列と同じ順序に従う必要があります。

   たとえば、xCharCounterUUID_TYPE がアクセスされたとき vReadCounter がトリガーされ、vWriteCommand がアクセスされたとき xCharControlUUID_TYPE がトリガーされる場合、配列を次のように定義します。

```
static const IotBleAttributeEventCallback_t pxCallBackArray[egattDemoNbAttributes]
 =
    {
  NULL,
  vReadCounter,
  vEnableNotification,
```

```
  vWriteCommand
};
```

5. サービスを作成します。

```
static const BTService_t xGattDemoService =
{
  .xNumberOfAttributes = egattDemoNbAttributes,
  .ucInstId = 0,
  .xType = eBTServiceTypePrimary,
  .pusHandlesBuffer = usHandlesBuffer,
  .pxBLEAttributes = (BTAttribute_t *)pxAttributeTable
};
```

6. 前のステップで作成した構造を使用して、API IotBle_CreateService を呼び出します。ミドルウェアは、すべてのサービスの作成に同期するため、IotBle_AddCustomServicesCb コールバックがトリガーされるとき、新しいサービスは既に定義されている必要があります。

   a. vendors/*vendor*/boards/*board*/aws_demos/config_files/iot_ble_config.h で IOT_BLE_ADD_CUSTOM_SERVICES を 1 に設定します。

   b. アプリケーションで IotBle_AddCustomServicesCb を作成します。

```
void IotBle_AddCustomServicesCb(void)
{
    BTStatus_t xStatus;
    /* Select the handle buffer. */
    xStatus = IotBle_CreateService( (BTService_t *)&xGattDemoService,
 (IotBleAttributeEventCallback_t *)pxCallBackArray );
}
```

移植

ユーザー入力および出力周辺機器

セキュアな接続には、数値比較のために入力と出力の両方が必要です。イベントマネージャを使用して eBLENumericComparisonCallback イベントを登録できます。

```
xEventCb.pxNumericComparisonCb = &prvNumericComparisonCb;
xStatus = BLE_RegisterEventCb( eBLENumericComparisonCallback, xEventCb );
```

周辺機器は数値のパスキーを表示し、比較の結果を入力として取得する必要があります。

API 実装の移植

FreeRTOS を新しいターゲットに移植するには、Wi-Fi プロビジョニングサービスおよび Bluetooth Low Energy 機能用にいくつかの API を実装する必要があります。

Bluetooth Low Energy API

FreeRTOS Bluetooth Low Energy ミドルウェアを使用するには、一部の API を実装する必要があります。

Bluetooth Classic 用 GAP と Bluetooth Low Energy 用 GAP の共通 API

- `pxBtManagerInit`
- `pxEnable`
- `pxDisable`
- `pxGetDeviceProperty`
- `pxSetDeviceProperty` (すべてのオプションは、`eBTpropertyRemoteRssi` と `eBTpropertyRemoteVersionInfo` を必須とします)
- `pxPair`
- `pxRemoveBond`
- `pxGetConnectionState`
- `pxPinReply`
- `pxSspReply`
- `pxGetTxpower`
- `pxGetLeAdapter`
- `pxDeviceStateChangedCb`
- `pxAdapterPropertiesCb`
- `pxSspRequestCb`
- `pxPairingStateChangedCb`
- `pxTxPowerCb`

Bluetooth Low Energy の GAP に固有の API

- `pxRegisterBleApp`
- `pxUnregisterBleApp`

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

GATT サーバー

- 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`

FreeRTOS Bluetooth Low Energy ライブラリをご使用のプラットフォームに移植する方法については、FreeRTOS 移植ガイドの Bluetooth Low Energy ライブラリの移植を参照してください。

FreeRTOS Bluetooth デバイス用の Mobile SDK

> **Important**
>
> このライブラリは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

FreeRTOS Bluetooth デバイス用 Mobile SDK を使用して、Bluetooth Low Energy 経由でマイクロコントローラーを操作するモバイルアプリケーションを作成できます。Mobile SDKs は、ユーザー認証に Amazon Cognito を使用して AWS サービスと通信することもできます。

FreeRTOS Bluetooth デバイス用の Android SDK

FreeRTOS Bluetooth デバイス用 Android SDK を使用して、Bluetooth Low Energy 経由でマイクロコントローラーを操作する Android モバイルアプリケーションを構築します。SDK は GitHub で入手できます。

FreeRTOS Bluetooth デバイス用 Android SDK をインストールするには、プロジェクトの README.md ファイルの「SDK のセットアップ」にある手順に従います。

SDK に付属しているデモモバイルアプリケーションのセットアップおよび実行の詳細については、「前提条件」および「FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション」を参照してください。

FreeRTOS Bluetooth デバイス用の iOS SDK

FreeRTOS Bluetooth デバイス用 iOS SDK を使用して、Bluetooth Low Energy 経由でマイクロコントローラーを操作する iOS モバイルアプリケーションを構築します。SDK は GitHub で入手できます。

iOS SDK をインストールするには

1. CocoaPods をインストールする:

```
$ gem install cocoapods
$ pod setup
```

> **Note**
>
> sudo を使用して CocoaPods をインストールする必要がある場合があります。

2. CocoaPods を使用して SDK をインストールします (これをポッドファイルに追加)。

```
$ pod 'FreeRTOS', :git => 'https://github.com/aws/amazon-freertos-ble-ios-sdk.git'
```

SDK に付属しているデモモバイルアプリケーションのセットアップおよび実行の詳細については、「前提条件」および「FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション」を参照してください。

付録 A: MQTT over BLE の GATT プロファイル

GATT サービスの詳細

MQTT over BLE は、データ転送 GATT サービスのインスタンスを使用して、FreeRTOS デバイスとプロキシデバイスの間で MQTT の簡潔なバイナリオブジェクト表現 (CBOR) メッセージを送信します。データ転送サービスでは、BLE GATT プロトコルを介した未加工データの送受信に役立つ特定

の特性が公開されています。また、BLE の最大伝送単位 (MTU) サイズを超えるペイロードのフラグメント化とアセンブリも処理します。

サービス UUID

A9D7-166A-D72E-40A9-A002-4804-4CC3-FF00

サービスインスタンス

ブローカーとの MQTT セッションごとに GATT サービスの 1 つのインスタンスが作成されます。各サービスには、そのタイプを識別する一意の UUID (2 バイト) があります。個々のインスタンスは、インスタンス ID によって区別されます。

各サービスは、各 BLE サーバーデバイス上でプライマリサービスとしてインスタンス化されます。特定のデバイス上に、サービスの複数のインスタンスを作成できます。MQTT プロキシサービスタイプには一意の UUID があります。

特性

特性コンテンツ形式: CBOR

最大特性値サイズ: 512 バイト

特徴	要件	必須のプロパティ	オプションのプロパティ	セキュリティアクセス許可	簡単な説明	UUID
コントロール	M	書き込み	なし	書き込みに暗号化が必要	MQTT プロキシを開始および停止するために使用されます。	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF01
TXMessage	M	読み取り、通知	なし	読み取りに暗号化が必要	メッセージを含む通知をプロキシ経由でブローカー	A9D7-166A-D72E-40A9-A002-48

特徴	要件	必須のプロパティ	オプションのプロパティ	セキュリティアクセス許可	簡単な説明	UUID
					に送信するために使用されます。	04-4CC3-FF02
RXMessage	M	応答なしの読み取り、書き込み	なし	読み取り、書き込みに暗号化が必要	プロキシ経由でブローカーからメッセージを受信するために使用されます。	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF03
TXLargeMessage	M	読み取り、通知	なし	読み取りに暗号化が必要	プロキシ経由でブローカーに大容量メッセージ (メッセージ > BLE MTU サイズ) を送信するために使用されます。	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF04

特徴	要件	必須のプロパティ	オプションのプロパティ	セキュリティアクセス許可	簡単な説明	UUID
RXLargeMessage	M	応答なしの読み取り、書き込み	なし	読み取り、書き込みに暗号化が必要	プロキシ経由でブローカーから大容量メッセージ (メッセージ > BLE MTU サイズ) を受信するために使用されます。	A9D7-166A-D72E-40A9-A002-4804-4CC3-FF05

GATT プロシージャの要件

特性値の読み取り	必須
長い特性値の読み取り	必須
特性値の書き込み	必須
長い特性値の書き込み	必須
特性記述子の読み取り	必須
特性記述子の書き込み	必須
通知	必須
表示	必須

メッセージタイプ

以下のメッセージタイプが交換されます。

メッセージの種類	メッセージ	マップに使用するキーと値のペア
0x01	CONNECT	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (1) • キー =「d」、値 = タイプ 3、テキスト文字列、セッションのクライアント ID • キー =「a」、値 = タイプ 3、テキスト文字列、セッションのブローカーエンドポイント • キー =「c」、値 = 単純値タイプ True/False
0x02	CONNACK	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (2) • キー =「s」、値 = タイプ 0 整数、ステータスコード
0x03	発行	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (3) • キー =「u」、値 = タイプ 3、テキスト文字列、パブリッシュ対象のトピック • キー =「n」、値 = タイプ 0、整数、パブリッシュの QoS

メッセージの種類	メッセージ	マップに使用するキーと値のペア
		• キー =「i」、値 = タイプ 0、整数、メッセージ識別子、QoS 1 パブリッシュのみ • キー =「k」、値 = タイプ 2、バイト文字列、パブリッシュのペイロード
0x04	PUBACK	• QoS 1 メッセージに対してのみ送信されます。 • キー =「w」、値 = タイプ 0 整数、メッセージタイプ (4) • キー =「i」、値 = タイプ 0、整数、メッセージ識別子
0x08	サブスクライブ	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (8) • キー =「v」、値 = タイプ 4、テキスト文字列の配列、サブスクリプション対象のトピック • キー =「o」、値 = タイプ 4、整数の配列、サブスクリプションの QoS • キー =「i」、値 = タイプ 0、整数、メッセージ識別子

メッセージの種類	メッセージ	マップに使用するキーと値のペア
0x09	SUBACK	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (9) • キー =「i」、値 = タイプ 0、整数、メッセージ識別子 • キー =「s」、値 = タイプ 0、整数、サブスクリプションのステータスコード
0X0A	UNSUBSCRIBE	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (10) • キー =「v」、値 = タイプ 4、テキスト文字列の配列、サブスクリプション解除対象のトピック • キー =「i」、値 = タイプ 0、整数、メッセージ識別子
0x0B	UNSUBACK	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (11) • キー =「i」、値 = タイプ 0、整数、メッセージ識別子 • キー =「s」、値 = タイプ 0、整数、サブスクリプション解除のステータスコード

メッセージの種類	メッセージ	マップに使用するキーと値のペア
0X0C	PINGREQ	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (12)
0x0D	PINGRESP	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (13)
0x0E	DISCONNNECT	• キー =「w」、値 = タイプ 0 整数、メッセージタイプ (14)

大容量ペイロード転送特性

TXLargeMessage

TXLargeMessage は、BLE 接続に対してネゴシエートされた MTU サイズより大きい大容量ペイロードを送信するために、デバイスによって使用されます。

- デバイスはペイロードの最初の MTU バイトを特性を介して通知として送信します。
- プロキシは、この特性に対して、残りのバイトの読み取りリクエストを送信します。
- デバイスは、MTU サイズまたはペイロードの残りのバイト数のいずれか小さい方のサイズを送信します。そのたびに、送信されたペイロードのサイズ分だけ読み取り済みのオフセットを増やします。
- プロキシは、取得したペイロードの長さが 0 になるか、ペイロードが MTU サイズより小さくなるまで、特性の読み取りを続けます。
- 指定されたタイムアウト内にデバイスが読み取りリクエストを受け取らない場合、転送は失敗し、プロキシとゲートウェイはバッファを解放します。
- 指定されたタイムアウト内にプロキシが読み取りレスポンスを受け取らない場合、転送は失敗し、プロキシはバッファを解放します。

RXLargeMessage

RXLargeMessage は、BLE 接続に対してネゴシエートされた MTU サイズより大きい大容量ペイロードを受信するためにデバイスによって使用されます。

- プロキシは、この特性に基づいてレスポンス付き書き込みを使用して、MTU サイズまでのメッセージを 1 つずつ書き込みます。
- デバイスは、長さがゼロ、または MTU サイズより小さい書き込みリクエストを受信するまでメッセージをバッファします。
- 指定されたタイムアウト内にデバイスが書き込みリクエストを受け取らない場合、転送は失敗し、デバイスはバッファを解放します。
- 指定されたタイムアウト内にプロキシが書き込みレスポンスを受け取らない場合、転送は失敗し、プロキシはバッファを解放します。

セルラーインターフェイスライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

セルラーインターフェイスライブラリは、セルラーモデム固有の AT コマンドの複雑さを隠し、ソケットのようなインターフェイスを C プログラマーに公開する、シンプルな統合 API を実装しています。

ほとんどのセルラーモデムには、3GPP TS v27.007 規格で定義されている AT コマンドが実装されています。このプロジェクトでは、再利用可能な共通コンポーネントに、このような標準 AT コマンドが実装されています。プロジェクトの 3 つのセルラーインターフェイスライブラリはすべて、その共通コードを利用しています。各モデムのライブラリでは、ベンダー固有の AT コマンドのみが実装され、完全なセルラーインターフェイスライブラリ API が公開されます。

3GPP TS v27.007 規格を実装する共通コンポーネントは、次のコード品質基準に準拠して記述されています。

- GNU Complexity スコアが 8 を超えない
- MISRA C:2012 コーディング標準。標準からの逸脱は、「coverity」でマークされたソースコードのコメントに記載されます。

依存関係と要件

セルラーインターフェイスライブラリに直接的な依存関係はありません。ただし、イーサネット、Wi-Fi、セルラーは FreeRTOS ネットワークスタックで共存できません。開発者は、ネットワークインターフェイスのいずれかを選択して、セキュアソケットライブラリと統合する必要があります。

移植

セルラーインターフェイスライブラリをご使用のプラットフォームに移植する方法については、FreeRTOS 移植ガイドのセルラーインターフェイスライブラリの移植を参照してください。

メモリ使用量

セルラーインターフェイスのコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
cellular_3gpp_api.c	6.3 K	5.7 K
cellular_3gpp_urc_handler.c	0.9 K	0.8 K
cellular_at_core.c	1.4 K	1.2 K
cellular_common_api.c	0.5 K	0.5 K
cellular_common.c	1.6 K	1.4 K
cellular_pkthandler.c	1.4 K	1.2 K
cellular_pktio.c	1.8 K	1.6 K
合計 (概算)	13.9 K	12.4 K

使用開始方法

ソースコードのダウンロード

ソースコードは、FreeRTOS ライブラリの一部として、または単独でダウンロードできます。

HTTPS を使用して GitHub からライブラリのクローンを作成するには

```
git clone https://github.com/FreeRTOS/FreeRTOS-Cellular-Interface.git
```

SSH を使用する場合

```
git clone git@github.com:FreeRTOS/FreeRTOS-Cellular-Interface.git
```

フォルダ構造

このリポジトリのルートには、以下のフォルダが表示されます。

- source: 3GPP TS v27.007 で定義されている標準 AT コマンドを実装する再利用可能な共通コード
- doc: ドキュメント
- test: 単体テストと cbmc
- tools: Coverity 静的解析と CMock 用のツール

ライブラリの設定とビルド

セルラーインターフェイスライブラリは、アプリケーションの一部として構築する必要があります。これを行うには、特定の設定を指定する必要があります。FreeRTOS_Cellular_Interface_Windows_Simulator プロジェクトには、ビルド設定の例があります。詳細については、セルラー API リファレンスを参照してください。

詳細については、セルラーインターフェイスページをを参照してください。

セルラーインターフェイスライブラリと MCU プラットフォームを統合する

セルラーインターフェイスライブラリは、抽象化されたインターフェイス (通信インターフェイス) を使用して MCU 上で実行され、セルラーモデムと通信します。通信インターフェイスは MCU プラットフォームにも実装する必要があります。通信インターフェイスの最も一般的な実装では UART ハードウェアを介して通信しますが、SPI などの他の物理インターフェイスを介しても実装できます。通信インターフェイスに関するドキュメントは、セルラーライブラリ API リファレンスにあります。通信インターフェイスの実装例を以下に示します。

- FreeRTOS Windows シミュレータ通信インターフェイス
- FreeRTOS 共通 IO UART 通信インターフェイス
- STM32 L475 ディスカバリーボード通信インターフェイス

- Sierra Sensor Hub ボード通信インタフェース

共通 I/O

> **Important**
>
> このライブラリは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

概要

一般に、デバイスドライバーは基盤となるオペレーティングシステムに依存せず、特定のハードウェア構成に固有です。ハードウェア抽象化レイヤー (HAL) は、ドライバーと上位レベルのアプリケーションコード間の共通のインターフェイスを提供します。HAL は、特定のドライバーの動作の詳細を抽象化し、そのようなデバイスを制御するための統一された API を提供します。同じ API を使用して、複数のマイクロコントローラー (MCU) ベースのリファレンスボード間でさまざまなデバイスドライバーにアクセスできます。

FreeRTOS 共通 I/O は、このハードウェア抽象化レイヤーとして機能します。サポートされているリファレンスボード上の共通シリアルデバイスにアクセスするための標準 API のセットを提供します。これらの共通 API は、これらの周辺機器と通信、相互作用し、プラットフォーム間でコードを機能させることができます。共通 I/O がない場合、低レベルデバイスで動作するコードの記述は、シリコンベンダー固有です。

サポートされている周辺機器

- UART
- SPI
- I2C

サポートされている機能

- 同期読み取り/書き込み: 要求された量のデータが転送されるまで、関数は戻りません。
- 非同期読み取り/書き込み: 関数はすぐに戻り、データ転送は非同期的に行われます。アクションが完了すると、登録されたユーザーコールバックが呼び出されます。

周辺機器固有

- I2C: 複数のオペレーションを 1 つのトランザクションに結合します。1 つのトランザクションで書き込みおよび読み取りアクションを実行するために使用されます。
- SPI: プライマリとセカンダリの間でデータを転送します。つまり、書き込みと読み取りが同時に行われます。

移植

詳細については、FreeRTOS 移植ガイドを参照してください。

AWS IoT Device Defender ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

AWS IoT Device Defender ライブラリを使用して、IoT デバイスから にセキュリティメトリクスを送信できます AWS IoT Device Defender。各デバイスに適切な動作として定義した値からの逸脱について、 AWS IoT Device Defender を使用してデバイスのセキュリティメトリクスを継続的にモニタリングできます。何かが正しく見えない場合、 AWS IoT Device Defender は問題を解決するためのアクションを実行できるようにアラートを送信します。とのやり取りでは、軽量のパブリッシュ/サブスクライブプロトコルである MQTT AWS IoT Device Defender を使用します。このライブラリは、AWS IoT Device Defenderで使用される MQTT トピック文字列を構成して認識するための API を提供します。

詳細については、「AWS IoT デベロッパーガイド」の「AWS IoT Device Defender」を参照してください。

ライブラリは C 言語で記述されており、ISO C90 と MISRA C:2012 に準拠するように設計されています。ライブラリには、標準 C ライブラリ以外のライブラリへの依存関係はありません。また、スレッディングや同期など、プラットフォームの依存関係もありません。任意の MQTT ライブラリと JSON または CBOR ライブラリで使用できます。ライブラリには、安全にメモリを使用し、ヒープ割り当てがないことを示すプルーフがあります。そのため、IoT マイクロコントローラーに適しています。また、他のプラットフォームに完全に移植することもできます。

AWS IoT Device Defender ライブラリは自由に使用でき、MIT オープンソースライセンスに基づいて配布されます。

AWS IoT Device Defender のコードサイズ (ARM Cortex-M 用の GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
defender.c	1.1 K	0.6 K
合計 (概算)	1.1 K	0.6 K

AWS IoT Greengrass 検出ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

概要

AWS IoT Greengrass 検出ライブラリは、お使いのマイクロコントローラーデバイスで、ネットワーク上の Greengrass コアを検出するために使用されます。 AWS IoT Greengrass 検出 API を使用して、デバイスは、コアのエンドポイントを見つけた後、Greengrass コアにメッセージを送信できます。

依存関係と要件

Greengrass Discovery ライブラリを使用するには、証明書とポリシー AWS IoTを含むモノを で作成する必要があります。詳細については、「AWS IoT の使用開始」を参照してください。

freertos/demos/include/aws_clientcredential.h ファイルで、以下の定数の値を設定する必要があります。

clientcredentialMQTT_BROKER_ENDPOINT

AWS IoT エンドポイント。

clientcredentialIOT_THING_NAME

IoT モノの名前。

clientcredentialWIFI_SSID

Wi-Fi ネットワークの SSID。

clientcredentialWIFI_PASSWORD

Wi-Fi のパスワード。

clientcredentialWIFI_SECURITY

Wi-Fi ネットワークで使用されるセキュリティの種類。

freertos/demos/include/aws_clientcredential_keys.h ファイルで、以下の定数の値も設定する必要があります。

keyCLIENT_CERTIFICATE_PEM

モノに関連付けられた PEM 証明書。

keyCLIENT_PRIVATE_KEY_PEM

モノに関連付けられているプライベートキー PEM。

コンソールに Greengrass グループとコアデバイスを設定する必要があります。詳細については、「 の開始方法 AWS IoT Greengrass」を参照してください。

coreMQTT ライブラリは Greengrass 接続には必要ありませんが、インストールすることを強くお勧めします。このライブラリは、検出された後、Greengrass コアと通信するために使用できます。

API リファレンス

完全な API リファレンスについては、Greengrass API リファレンスを参照してください。

使用例

Greengrass ワークフロー

MCU デバイスは、Greengrass コア接続パラメータを含む JSON ファイルから をリクエストすることで AWS IoT 、検出プロセスを開始します。JSON ファイルから Greengrass コア接続パラメータを取得する方法は 2 つあります。

- 自動選択は、JSON ファイルにリストされているすべての Greengrass コアを反復処理し、使用可能な最初のコアに接続します。

- 手動選択では、aws_ggd_config.h の情報を使用して指定された Greengrass コアに接続します。

Greengrass API の使用方法

Greengrass API のすべてのデフォルト設定オプションは aws_ggd_config_defaults.h で定義されています。

Greengrass のコアが 1 つだけの場合、GGD_GetGGCIPandCertificate を呼び出して、Greengrass コア接続情報で JSON ファイルを要求します。GGD_GetGGCIPandCertificate が返されると、pcBuffer パラメータに JSON ファイルのテキストが格納されます。pxHostAddressData パラメータには、接続可能な Greengrass コアの IP アドレスとポートが含まれます。

動的に証明書を割り当てるなどのカスタマイズオプションを追加するには、次の API を呼び出す必要があります。

GGD_JSONRequestStart

Greengrass コアを検出する検出リクエストを開始する AWS IoT HTTP GET リクエストを に送ります。 GD_SecureConnect_Sendはリクエストの送信先として使用されます AWS IoT。

GGD_JSONRequestGetSize

HTTP レスポンスから JSON ファイルのサイズを取得します。

GGD_JSONRequestGetFile

JSON オブジェクト文字列を取得します。GGD_JSONRequestGetSize と GGD_JSONRequestGetFile は GGD_SecureConnect_Read を使用してソケットから JSON データを取得します。 AWS IoTから JSON データを受け取るには、GGD_JSONRequestStart、GGD_SecureConnect_Send、GGD_JSONRequestGetSize を呼び出す必要があります。

GGD_GetIPandCertificateFromJSON

JSON データから IP アドレスと Greengrass コア証明書を抽出します。xAutoSelectFlag を True に設定すると、自動選択をオンにすることができます。自動選択は、FreeRTOS デバイスが接続できる最初のコアデバイスを検出します。Greengrass コアに接続するには、GGD_SecureConnect_Connect 関数を呼び出し、コアデバイスの IP アドレス、ポート、および証明書を渡します。手動選択を使用するには、HostParameters_t パラメータの次のフィールドを設定します。

`pcGroupName`

コアが属する Greengrass グループの ID。`aws greengrass list-groups` CLI コマンドを使用して、Greengrass グループの ID を見つけることができます。

`pcCoreAddress`

接続している Greengrass コアの ARN。

coreHTTP ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

小型 IoT デバイス (MCU または小型 MPU) 用の HTTP C クライアントライブラリ

序章

coreHTTP ライブラリは、HTTP/1.1 標準のサブセットのクライアント実装です。HTTP 標準は、TCP/IP 上で動作するステートレスプロトコルを提供し、分散型かつ協調型のハイパーテキスト情報システムでよく使用されます。

coreHTTP ライブラリは、HTTP/1.1 プロトコル標準のサブセットを実装しています。このライブラリは、低メモリフットプリントに最適化されています。ライブラリは完全同期 API を提供するため、アプリケーションは並行処理を完全に管理できます。固定バッファのみを使用するため、アプリケーションはメモリ割り当て方式を完全に制御できます。

ライブラリは C 言語で記述されており、ISO C90 と MISRA C:2012 に準拠するように設計されています。ライブラリの唯一の依存関係は標準 C ライブラリと Node.js の http-parser の LTS バージョン (v12.19.1) です。ライブラリには、安全にメモリを使用し、ヒープ割り当てがないことを示すプルーフがあります。そのため、IoT マイクロコントローラーに適しています。また、他のプラットフォームに完全に移植することもできます。

IoT アプリケーションで HTTP 接続を使用する場合は、coreHTTP 相互認証のデモに示すような TLS プロトコルを使用するインターフェイスなど、セキュアなトランスポートインターフェイスを使用することをお勧めします。

このライブラリは無償で使用でき、MIT オープンソースライセンスに基づいて配布されます。

coreHTTP のコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
core_http_client.c	3.2 K	2.6 K
api.c (llhttp)	2.6 K	2.0 K
http.c (llhttp)	0.3 K	0.3 K
llhttp.c (llhttp)	17.9	15.9
合計 (概算)	23.9K	20.7K

coreJSON ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

JSON (JavaScript Object Notation) は、人間が読解可能なデータシリアル化形式です。JSON は AWS IoT Device Shadow などのサービスでデータ交換に広く使用されており、GitHub REST API など多くの API に含まれています。JSON は Ecma International によって標準として管理されています。

coreJSON ライブラリは、ECMA-404 標準 JSON データ交換構文を厳密に適用し、キー検索をサポートするパーサーを提供します。ライブラリは C 言語で記述されており、ISO C90 と MISRA C:2012 に準拠するように設計されています。ライブラリには、安全にメモリを使用し、ヒープ割り当てがないことを示すプルーフがあります。そのため、IoT マイクロコントローラーに適しています。また、他のプラットフォームに完全に移植することもできます。

メモリ使用量

coreJSON ライブラリは、内部スタックを使用して JSON ドキュメント内のネストされた構造をトラッキングします。スタックは 1 つの関数呼び出しの間存在し、保持されません。スタックサイズ

は、マクロ (JSON_MAX_DEPTH) を定義することで指定できます。デフォルトは 32 レベルです。各レベルは 1 バイトを消費します。

coreJSON のコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
core_json.c	2.9 K	2.4 K
合計 (概算)	2.9 K	2.4 K

coreMQTT ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

coreMQTT ライブラリは、MQTT (Message Queue Telemetry Transport) 標準のクライアント実装です。MQTT 標準は軽量な公開/サブスクライブ (または PubSub) メッセージングプロトコルを提供します。これは TCP/IP 上で動作し、マシン間 (M2M) および IoT ユースケースでよく使用されます。

coreMQTT ライブラリは、MQTT 3.1.1 プロトコル標準に準拠しています。このライブラリは、低メモリフットプリントに最適化されています。このライブラリの設計では、QoS 0 MQTT PUBLISH メッセージのみを使用するリソース制約のあるプラットフォームから、QoS 2 MQTT PUBLISH over TLS (Transport Layer Security) 接続を使用するリソース豊富なプラットフォームまで、さまざまなユースケースを取り入れています。ライブラリには組み合わせ可能な関数メニューが用意されており、特定のユースケースのニーズに正確に適合するように選択して組み合わせることができます。

ライブラリは C 言語で記述されており、ISO C90 と MISRA C:2012 に準拠するように設計されています。この MQTT ライブラリは、以下を除くライブラリには依存しません。

- 標準 C ライブラリ
- お客様が実装したネットワークトランスポートインターフェイス
- (オプション) ユーザーが実装したプラットフォーム時間関数

ライブラリは、単純な送受信トランスポートインターフェイス仕様を提供することにより、基盤となるネットワークドライバーから疎結合化されます。アプリケーションライターは、既存のトランスポートインターフェイスを選択したり、アプリケーションに応じて独自のトランスポートインターフェイスを実装したりできます。

ライブラリは、MQTT ブローカーへの接続、トピックへのサブスクライブ/サブスクライブ解除、トピックへのメッセージの公開、受信メッセージの受信を行うための高レベル API を提供します。この API は、上記のトランスポートインターフェイスをパラメーターとして受け取り、それを使用して MQTT ブローカとの間でメッセージを送受信します。

ライブラリは、低レベルのシリアライザ/デシリアライザ API も公開しています。この API を使用すると、他のオーバーヘッドなしで、必要なサブセットの MQTT 機能のみで構成されるシンプルな IoT アプリケーションを構築できます。シリアライザ/デシリアライザ API は、ソケットなどの利用可能なトランスポートレイヤー API と組み合わせて使用して、ブローカとの間でメッセージを送受信できます。

IoT アプリケーションで MQTT 接続を使用する場合は、TLS プロトコルを使用するインターフェイスなど、セキュアなトランスポートインターフェイスを使用することをお勧めします。

この MQTT ライブラリには、スレッディングや同期など、プラットフォームの依存関係はありません。このライブラリには、安全にメモリを使用し、ヒープ割り当てがないことを示すプルーフがあります。そのため、IoT マイクロコントローラーに適しています。また、他のプラットフォームに完全に移植することもできます。無償で使用でき、MIT オープンソースライセンスに基づいて配布されます。

coreMQTT のコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
core_mqtt.c	4.0K	3.4K
core_mqtt_state.c	1.7 K	1.3K
core_mqtt_serializer.c	2.8K	2.2 K
合計 (概算)	8.5K	6.9 K

coreMQTT エージェントライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

coreMQTT エージェントライブラリは、スレッドセーフを coreMQTT ライブラリ に追加する高レベル API です。これにより、バックグラウンドで MQTT 接続を管理し、他のタスクからの介入を必要としない専用の MQTT エージェントタスクを作成できます。このライブラリでは coreMQTT の API と同等のスレッドセーフを提供するため、マルチスレッド環境で使用できます。

MQTT エージェントは、独立したタスク (または実行スレッド) です。MQTT ライブラリの API へのアクセスが許可される唯一のタスクであるため、スレッドセーフが実現されています。すべての MQTT API 呼び出しを単一のタスクに分離することにより、アクセスはシリアル化されます。そのため、セマフォやその他の同期プリミティブが必要ありません。

このライブラリでは、スレッドセーフなメッセージングキュー (または他のプロセス間通信メカニズム) を使用して、MQTT API を呼び出すすべてのリクエストをシリアル化します。メッセージングの実装は、メッセージングインターフェイスを介してライブラリから切り離されるため、ライブラリを他のオペレーティングシステムに移植できます。メッセージングインターフェイスは、エージェントのコマンド構造へのポインタを送受信する関数と、これらのコマンドオブジェクトを割り当てる関数で構成されています。これにより、アプリケーションの作成者はアプリケーションに適したメモリ割り当て方式を決定できます。

ライブラリは C 言語で記述されており、ISO C90 と MISRA C:2012 に準拠するように設計されています。ライブラリには、coreMQTT ライブラリと標準 C ライブラリ以外のライブラリへの依存関係はありません。ライブラリには、安全にメモリを使用し、ヒープ割り当てがないことを示すプルーフがあります。そのため、IoT マイクロコントローラーに使用できます。また、他のプラットフォームに移植可能です。

このライブラリは無償で使用でき、MIT オープンソースライセンスに基づいて配布されます。

coreMQTT エージェントのコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
core_mqtt_agent.c	1.7 K	1.5 K
core_mqtt_agent_command_functions.c	0.3 K	0.2 K
core_mqtt.c (coreMQTT)	4.0K	3.4K
core_mqtt_state.c (coreMQTT)	1.7 K	1.3K
core_mqtt_serializer.c (coreMQTT)	2.8K	2.2 K
合計 (概算)	10.5K	8.6K

AWS IoT 無線通信経由 (OTA) ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

AWS IoT 無線通信 (OTA) 更新ライブラリを使用すると、プロトコルとして HTTP または MQTT を使用する FreeRTOS デバイスのファームウェア更新の通知、ダウンロード、検証を管理できます。OTA エージェントライブラリを使用すると、ファームウェア更新とデバイス上で実行されているアプリケーションを論理的に分離することができます。OTA エージェントは、アプリケーションとネットワーク接続を共有できます。ネットワーク接続を共有することで、大量の RAM を節約できます。さらに、OTA エージェントライブラリを使用すると、ファームウェア更新をテスト、コミット、ロールバックするためのアプリケーション固有のロジックを定義できます。

IoT によって、従来は接続されていなかった組み込みデバイスにインターネット接続機能が導入されました。これらのデバイスは、インターネットで使用可能なデータを通信するようにプログラムで

き、リモートでモニタリングや制御ができます。テクノロジーの進歩により、コンシューマー、産業、およびエンタープライズ分野でこれら従来の組み込みデバイスにインターネット機能が急速に普及しています。

IoT デバイスは、通常、大量にデプロイされ、多くの場合人間のオペレータがアクセスすることが困難または実用的でない場所に配置されます。データを漏洩する可能性のあるセキュリティ脆弱性が発見されるシナリオを想像してみてください。このようなシナリオでは、影響を受けるデバイスをセキュリティ修正で迅速かつ確実に更新することが重要です。OTA 更新を実行できないと、地理的に分散しているデバイスの更新も困難になる可能性があります。技術者にこれらのデバイスを更新してもらうことは、コストがかかり、時間がかかり、しばしば実用的ではありません。これらのデバイスの更新に時間がかかるため、セキュリティ脆弱性に長期間さらされます。更新のためにこれらのデバイスをリコールすることもコストがかかり、ダウンタイムによってコンシューマーに重大なサービスの中断が発生する可能性があります。

無線通信経由 (OTA) 更新を使用すると、費用のかかるリコールや技術者の派遣なしに、デバイスのファームウェアを更新できます。この方法には次の利点があります。

- セキュリティ: デバイスが現場にデプロイされた後に検出されたセキュリティ脆弱性やソフトウェアバグに迅速に対応できます。
- イノベーション: 新機能が開発されるたびに頻繁に製品を更新できるため、イノベーションサイクルを促進できます。更新は、従来の更新方法と比較して、最小限のダウンタイムで迅速に有効になります。
- コスト: OTA 更新によって、デバイスの更新に従来使用されていた方法と比較して、メンテナンスコストを大幅に削減できます。

OTA 機能を提供するには、次の設計上の考慮事項が必要です。

- 安全な通信: 更新では、転送中にダウンロードが改ざんされないように、暗号化された通信チャネルを使用する必要があります。
- 復旧: 断続的なネットワーク接続や無効な更新の受信などにより、更新が失敗することがあります。これらのシナリオでは、デバイスが安定した状態に復旧し、使用不能な状態にならないようにする必要があります。
- 作成者検証: バージョンや互換性の確認など、その他の検証とともに、信頼できるソースからの更新かどうかを検証する必要があります。

FreeRTOS での OTA 更新の設定の詳細については、「FreeRTOS 無線通信経由更新」を参照してください。

AWS IoT 無線通信経由 (OTA) ライブラリ

AWS IoT OTA ライブラリを使用すると、新しく利用可能な更新の通知を管理したり、ダウンロードしたり、ファームウェア更新の暗号化検証を実行したりできます。無線通信経由 (OTA) クライアントライブラリを使用して、デバイスで実行されているアプリケーションからファームウェア更新メカニズムを論理的に分離できます。無線通信経由 (OTA) クライアントライブラリは、アプリケーションとネットワーク接続を共有できるため、リソースに制約のあるデバイスのメモリを節約できます。また、無線通信経由 (OTA) クライアントライブラリを使用すると、ファームウェア更新をテスト、コミット、ロールバックするためのアプリケーション固有のロジックを定義できます。このライブラリは、Message Queuing Telemetry Transport (MQTT) や Hypertext Transfer Protocol (HTTP) などのさまざまなアプリケーションプロトコルをサポートし、ネットワークの種類と条件に合わせて微調整できる各種設定オプションを提供します。

このライブラリの API は、次の主要な機能を提供します。

- 通知を登録するか、利用可能な新しい更新リクエストをポーリングします。
- 更新リクエストを受け取り、解析し、検証します。
- 更新リクエストの情報に従って、ファイルをダウンロードして検証します。
- 受信した更新をアクティブ化する前にセルフテストを実行して、更新の機能的妥当性を確認します。
- デバイスのステータスを更新します。

このライブラリは、 AWS サービスを使用して、ファームウェア更新の送信、複数のリージョンにまたがる多数のデバイスのモニタリング、デプロイの障害範囲の縮小、更新のセキュリティの検証など、さまざまなクラウド関連機能を管理します。このライブラリは、任意の MQTT または HTTP ライブラリで使用できます。

このライブラリのデモでは、FreeRTOS デバイスで coreMQTT ライブラリと AWS サービスを使用して無線通信経由更新を完了します。

機能

OTA エージェントの各インターフェイスは次のとおりです。

`OTA_Init`

システムで OTA エージェント (「OTA タスク」) を起動して OTA エンジンを初期化します。OTA エージェントは 1 つしか存在しません。

OTA_Shutdown

OTA エージェントをシャットダウンする通知。OTA エージェントは、必要に応じてすべての MQTT ジョブ通知トピックのサブスクライブを解除し、進行中の OTA ジョブがあれば停止し、すべてのリソースをクリアします。

OTA_GetState

OTA エージェントの現在の状態を取得します。

OTA_ActivateNewImage

OTA を介して受信した最新のマイクロコントローラーファームウェアイメージを有効にします。(詳細なジョブのステータスはセルフテストになっているはずです)

OTA_SetImageState

現在実行中のマイクロコントローラーファームウェアイメージの検証状態 (テスト中、受け入れ済または拒否済) を設定します 。

OTA_GetImageState

現在実行中のマイクロコントローラーファームウェアイメージの状態 (テスト中、受け入れ済または拒否済) を取得します 。

OTA_CheckForUpdate

OTA 更新サービスから利用可能な次の OTA 更新を要求します。

OTA_Suspend

OTA エージェントのすべてのオペレーションを一時停止します。

OTA_Resume

OTA エージェントのオペレーションを再開します。

OTA_SignalEvent

OTA エージェントタスクにイベントを通知します。

OTA_EventProcessingTask

OTA エージェントイベント処理ループ。

OTA_GetStatistics

受信、キューイング、処理、ドロップされたパケットの数を含む OTA メッセージパケットの統計情報を取得します。

OTA_Err_strerror

OTA エラーのエラーコードから文字列への変換。

OTA_JobParse_strerror

OTA ジョブ解析エラーコードを文字列に変換します。

OTA_PalStatus_strerror

OTA PAL ステータスのステータスコードから文字列への変換。

OTA_OsStatus_strerror

OTA OS ステータスのステータスコードから文字列への変換。

API リファレンス

詳細については、「AWS IoT Over-the-air Update: Functions」を参照してください。

使用例

MQTT プロトコルを使用する一般的な OTA 対応デバイスアプリケーションは、次の一連の API コールを使用して OTA エージェントを起動します。

1. AWS IoT coreMQTT エージェントに接続します。詳細については、「coreMQTT エージェントライブラリ」を参照してください。
2. `OTA_Init` を呼び出すことによって、OTA エージェントを初期化します。バッファ、必要な ota インターフェイス、モノの名前、およびアプリケーションコールバックが含まれます。コールバックは、OTA 更新ジョブの完了後に実行されるアプリケーション固有のロジックを実装します。
3. OTA 更新が完了すると、FreeRTOS は、ジョブ完了コールバックを `accepted`、`rejected`、または `self test` のいずれかのイベントで呼び出します。
4. 新しいファームウェアイメージが拒否された場合 (たとえば検証エラーのため)、アプリケーションは通常、通知を無視して次の更新を待機します。
5. アップデートが有効で、承諾済みとマークされている場合は、`OTA_ActivateNewImage` を呼び出してデバイスをリセットし、新しいファームウェアイメージを起動します。

移植

プラットフォームに OTA 機能を移植する方法については、FreeRTOS 移植ガイドの OTA ライブラリの移植を参照してください。

メモリ使用量

AWS IoT OTA のコードサイズ (ARM Cortex-M 用の GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
ota.c	8.3K	7.5 K
ota_interface.c	0.1 K	0.1 K
ota_base64.c	0.6 K	0.6 K
ota_mqtt.c	2.4 K	2.2 K
ota_cbor.c	0.8 K	0.6 K
ota_http.c	0.3 K	0.3 K
合計 (概算)	12.5K	11.3K

corePKCS11 ライブラリ

> **Note**
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

概要

公開鍵暗号標準 #11 では、暗号化トークンを管理および使用するためのプラットフォームに依存しない API が定義されています。PKCS #11 は、標準で定義された API と、標準自体を指します。PKCS #11 暗号化 API は、キーストレージ、暗号化オブジェクトプロパティの取得/設定、およびセッションセマンティクスを抽象化します。この API は、一般的な暗号化オブジェクトの操作に

広く使用されており、重要です。この API で指定する関数により、アプリケーションソフトウェアは暗号化オブジェクトの使用、作成、変更、削除を、アプリケーションのメモリに公開することなく実行できるからです。例えば、FreeRTOS AWS リファレンス統合は PKCS #11 API の小さなサブセットを使用して、アプリケーションがキーを「見る」ことなく Transport Layer Security (TLS) プロトコルによって認証および保護されるネットワーク接続を作成するために必要なシークレット (プライベート) キーにアクセスします。

corePKCS11 ライブラリには、Mbed TLS が提供する暗号化機能を使用する PKCS #11 インターフェイス (API) のソフトウェアベースのモック実装が含まれています。ソフトウェアモックを使用すると、迅速な開発と柔軟性が実現できますが、このモックを実稼働用デバイスで使用されるセキュアキーストレージ固有の実装に置き換える必要があります。一般に、トラステッドプラットフォームモジュール (TPM)、ハードウェアセキュリティモジュール (HSM)、セキュアエレメント、またはその他のタイプのセキュアハードウェアエンクレーブなどのセキュアな暗号化プロセッサのベンダーは、PKCS #11 実装をハードウェアとともに配布します。したがって、corePKCS11 ソフトウェア専用モックライブラリの目的は、実稼働デバイスで暗号化プロセッサ固有の PKCS #11 実装に切り替える前に、迅速なプロトタイピングと開発ができるハードウェア専用ではない PKCS #11 実装を提供することです。

PKCS #11 標準のサブセットのみが実装されており、非対称キー、乱数生成、ハッシュなどの操作に重点が置かれています。対象となるユースケースには、小型の組み込みデバイスにおける TLS 認証に対応した証明書とキーの管理、およびコードサイン署名の検証などがあります。FreeRTOS ソースコードリポジトリのファイル `pkcs11.h` (標準的な本体である OASIS から入手) を参照してください。FreeRTOS リファレンス実装では、`SOCKETS_Connect` 中に TLS クライアントの認証を実行するために、PKCS #11 API コールが TLS ヘルパーインターフェイスによって使用されます。PKCS #11 API コールは、ワンタイムの開発者プロビジョニングワークフローによって、 AWS IoT MQTT ブローカーへの認証用 TLS クライアント証明書とプライベートキーをインポートするためにも使用されます。プロビジョニングと TLS クライアント認証の 2 つのユースケースでは、PKCS #11 インターフェイス規格の小さなサブセットのみを実装する必要があります。

機能

次の PKCS #11 のサブセットが使用されます。このリストは、プロビジョニング、TLS クライアント認証、およびクリーンアップをサポートするためにルーチンが呼び出される順序とほぼ同じです。これらの機能の詳細については、標準化組織が提供する PKCS #11 のドキュメントを参照してください。

一般的なセットアップとティアダウン API

- `C_Initialize`

- C_Finalize
- C_GetFunctionList
- C_GetSlotList
- C_GetTokenInfo
- C_OpenSession
- C_CloseSession
- C_Login

プロビジョニング API

- C_CreateObject CKO_PRIVATE_KEY (デバイスプライベートキー用)
- C_CreateObject CKO_CERTIFICATE (デバイス証明書とコード検証証明書用)
- C_GenerateKeyPair
- C_DestroyObject

クライアント承認

- C_GetAttributeValue
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_GenerateRandom
- C_SignInit
- C_Sign
- C_VerifyInit
- C_Verify
- C_DigestInit
- C_DigestUpdate
- C_DigestFinal

非対称暗号化方式のサポート

FreeRTOS リファレンス実装では、NIST P-256 曲線を使用する PKCS #11 2048 ビット RSA (署名のみ) と ECDSA をサポートしています。次の手順では、P-256 クライアント証明書に基づいて AWS IoT モノを作成する方法について説明します。

AWS CLI および OpenSSL の次の (またはより新しい) バージョンを使用していることを確認します。

```
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
```

次の手順では、`aws configure` コマンドを使用して AWS CLIを設定済みとします。詳細については、AWS Command Line Interface ユーザーガイドの aws configure を使用したクイック設定を参照してください。

P-256 クライアント証明書に基づいて AWS IoT モノを作成するには

1. AWS IoT モノを作成します。

   ```
   aws iot create-thing --thing-name thing-name
   ```

2. OpenSSL を使用して P-256 キーを作成します。

   ```
   openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt
    ec_param_enc:named_curve -outform PEM -out thing-name.key
   ```

3. ステップ 2 で作成したキーで署名された証明書の登録リクエストを作成します。

   ```
   openssl req -new -nodes -days 365 -key thing-name.key -out thing-name.req
   ```

4. 証明書の登録リクエストを に送信します 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. (前のコマンドで ARN 出力によって参照される) 証明書をモノにアタッチします。

```
aws iot attach-thing-principal --thing-name thing-name \
  --principal "arn:aws:iot:us-
east-1:123456789012:cert/
86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de729"
```

6. ポリシーを作成します。(このポリシーは許容範囲が非常に広いので、 開発目的でのみ使用してください。)

```
aws iot create-policy --policy-name FullControl --policy-document file://
policy.json
```

`create-policy` コマンドで指定された policy.json ファイルのリストを次に示します。Greengrass 接続と検出のための FreeRTOS デモを実行しない場合は、`greengrass:*` アクションを省略できます。

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

7. プリンシパル (証明書) とポリシーをモノにアタッチします。

```
aws iot attach-principal-policy --policy-name FullControl \
  --principal "arn:aws:iot:us-
east-1:123456789012:cert/
86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de729"
```

次に、このガイドの「AWS IoT の使用開始」セクションの手順に従います。作成した証明書とプライベートキーを aws_clientcredential_keys.h ファイルにコピーすることを忘れないでください。モノの名前は aws_clientcredential.h にコピーします。

> **Note**
>
> 証明書とプライベートキーは、デモ専用にハードコードされています。本番稼動レベルのアプリケーションでは、これらのファイルを安全な場所に保存する必要があります。

移植

プラットフォームに corePKCS11 ライブラリを移植する方法については、FreeRTOS 移植ガイドの corePKCS11 ライブラリの移植を参照してください。

メモリ使用量

corePKCS11 のコードサイズ (ARM Cortex-M 向けの GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
core_pkcs11.c	0.8 K	0.8 K
core_pki_utils.c	0.5 K	0.3 K
core_pkcs11_mbedtls.c	8.9 K	7.5 K
合計 (概算)	10.2K	8.6K

セキュアソケットライブラリ

> **Important**
>
> このライブラリは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

概要

FreeRTOS セキュアソケットライブラリを使用すると、安全に通信できる組み込みアプリケーションを作成できます。このライブラリは、さまざまなネットワークプログラミングの経歴を持つソフトウェア開発者が簡単にオンボードを行えるように設計されています。

FreeRTOS セキュアソケットライブラリは、バークレーソケットインターフェイスをベースにしています。また、TLS プロトコルによる安全な通信オプションも利用できます。FreeRTOS セキュアソケットライブラリとバークレーソケットインターフェイスの相違点の詳細については、「セキュアソケット API リファレンス」の「`SOCKETS_SetSockOpt`」を参照してください。

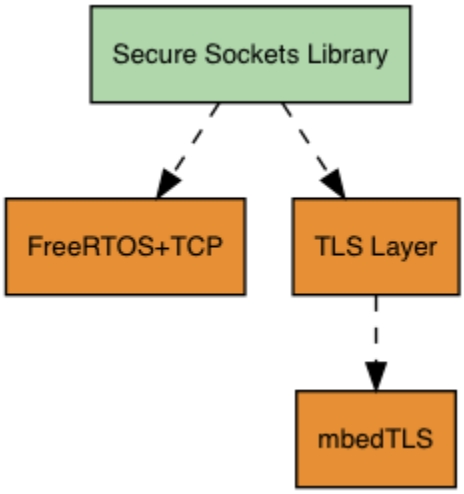
> **Note**
>
> 現在のところ FreeRTOS セキュアソケットでサポートされているのはクライアント API に加え、サーバー側 `Bind` API の lightweight IP (lwIP) 実装です。

依存関係と要件

FreeRTOS セキュアソケットライブラリは、TCP/IP スタックおよび TLS 実装に依存します。FreeRTOS のポートは、次の 3 つのいずれかの方法でこれらの依存関係に対応します。

- TCP/IP と TLS の両方のカスタム実装
- TCP/IP のカスタム実装、および mbedTLS を使用する FreeRTOS TLS レイヤー
- mbedTLS を使用する FreeRTOS+TCP および FreeRTOS TLS レイヤー

以下の依存関係の図は、FreeRTOS セキュアソケットライブラリに含まれるリファレンス実装を示しています。このリファレンス実装では、依存関係がある FreeRTOS+TCP および mbedTLS を使用して、TLS と TCP/IP over Ethernet および Wi-Fi がサポートされています。FreeRTOS TLS レイヤーの詳細については、「Transport Layer Security」を参照してください。



機能

FreeRTOS セキュアソケットライブラリの機能には次のものがあります。

- 標準のバークレーソケットベースのインターフェイス
- データの送受信のためのスレッドセーフな API
- TLS を簡単に有効化できる

トラブルシューティング

エラーコード

FreeRTOS セキュアソケットライブラリが返すエラーコードは負の値です。各エラーコードの詳細については、セキュアソケット API リファレンスの「セキュアソケットのエラーコード」を参照してください。

> **Note**
>
> FreeRTOS セキュアソケット API がエラーコードを返した場合、FreeRTOS セキュアソケットライブラリに依存している coreMQTT ライブラリ は、`AWS_IOT_MQTT_SEND_ERROR` というエラーコードを返します。

開発者サポート

FreeRTOS セキュアソケットライブラリには、IP アドレス処理用の 2 つのヘルパーマクロが含まれています。

SOCKETS_inet_addr_quick

このマクロは、4 つの別個のオクテットで表現された IP アドレスを、ネットワークバイト順に 32 ビットの数値で表現された IP アドレスに変換します。

SOCKETS_inet_ntoa

このマクロは、ネットワークバイト順に 32 ビットの数値で表現された IP アドレスを、ドット区切りの 10 進数表記の文字列に変換します。

使用制限

FreeRTOS セキュアソケットライブラリによってサポートされているのは TCP ソケットのみです。UDP ソケットはサポートされていません。

サーバー API は、サーバー側 Bind API の lightweight IP (lwIP) 実装を除いて FreeRTOS セキュアソケットライブラリではサポートされていません。クライアント API がサポートされています。

初期化

FreeRTOS セキュアソケットライブラリを使用するには、ライブラリおよびその依存関係を初期化する必要があります。セキュアソケットライブラリを初期化するには、アプリケーションで以下のコードを使用します。

```
BaseType_t xResult = pdPASS;
xResult = SOCKETS_Init();
```

依存ライブラリは個別に初期化する必要があります。たとえば、FreeRTOS+TCP が依存関係にある場合、アプリケーションで `FreeRTOS_IPInit` も呼び出す必要があります。

API リファレンス

完全な API リファレンスについては、「Secure Sockets API Reference」を参照してください。

使用例

以下のコードは、クライアントをサーバーに接続します。

```
#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 );
}
```

完全な例については、「セキュアソケットエコークライアントのデモ」を参照してください。

移植

FreeRTOS セキュアソケットは、TCP/IP スタックおよび TLS 実装に依存します。スタックによっては、セキュアソケットライブラリを移植するには、以下のいくつかの移植が必要になる場合があります。

- FreeRTOS+TCP TCP/IP スタック
- corePKCS11 ライブラリ
- Transport Layer Security

移植の詳細については、FreeRTOS 移植ガイドのセキュアソケットライブラリの移植を参照してください。

AWS IoT Device Shadow ライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

AWS IoT Device Shadow ライブラリを使用して、登録されたすべてのデバイスの現在の状態 (シャドウ) を保存および取得できます。デバイスのシャドウは、デバイスがオフラインであってもウェブアプリケーションで操作できる、デバイスの永続的な仮想表現です。デバイスの状態は、JSON ドキュメントにシャドウとしてキャプチャされます。MQTT または HTTP 経由で AWS IoT Device Shadow サービスにコマンドを送信して、最新の既知のデバイス状態をクエリしたり、状態を変更したりできます。各デバイスのシャドウは、対応するモノ (AWS クラウド上の特定のデバイスまたは論理エンティティの表現) の名前で一意に識別されます。詳細については、AWS IoTを使用したデバイスの管理を参照してください。シャドウの詳細については、AWS IoT のドキュメントを参照してください。

AWS IoT Device Shadow ライブラリには、標準 C ライブラリ以外のライブラリへの依存関係はありません。また、スレッディングや同期など、プラットフォームの依存関係もありません。任意の MQTT ライブラリと JSON ライブラリで使用できます。

このライブラリは無償で使用でき、MIT オープンソースライセンスに基づいて配布されます。

AWS IoT デバイスシャドウのコードサイズ (ARM Cortex-M 用の GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
shadow.c	1.2 K	0.9 K
合計 (概算)	1.2 K	0.9 K

AWS IoT ジョブライブラリ

> **Note**
>
> このページのコンテンツは最新ではない可能性があります。最新の更新については、FreeRTOS.org ライブラリのページを参照してください。

序章

AWS IoT ジョブは、1 つ以上の接続されたデバイスに保留中のジョブを通知するサービスです。ジョブを使用して、多数のデバイスの管理、デバイス上のファームウェアおよびセキュリティ証明書の更新、デバイスの再起動や診断の実行などの管理タスクの実行ができます。詳細については、AWS IoT デベロッパーガイドのジョブを参照してください。 AWS IoT Jobs サービスとのインタラクションでは、軽量のパブリッシュ/サブスクライブプロトコルである MQTT を使用します。このライブラリは、 AWS IoT ジョブサービスで使用される MQTT トピック文字列を構成および認識するための API を提供します。

AWS IoT Jobs ライブラリは C で記述され、ISO C90 および MISRA C:2012 に準拠するように設計されています。ライブラリには、標準 C ライブラリ以外のライブラリへの依存関係はありません。任意の MQTT ライブラリと JSON ライブラリで使用できます。ライブラリには、安全にメモリを使用し、ヒープ割り当てがないことを示すプルーフがあります。そのため、IoT マイクロコントローラーに適しています。また、他のプラットフォームに完全に移植することもできます。

このライブラリは無償で使用でき、MIT オープンソースライセンスに基づいて配布されます。

AWS IoT ジョブのコードサイズ (ARM Cortex-M 用の GCC で生成された例)		
ファイル	-O1 最適化を使用	-Os 最適化を使用
jobs.c	1.9K	1.6K
合計 (概算)	1.9K	1.6K

Transport Layer Security

> **Important**
>
> このライブラリは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

FreeRTOS Transport Layer Security (TLS) インターフェイスは、暗号実装の詳細をプロトコルスタックの上位の Secure Sockets Layer (SSL) インターフェイスから抽象化するために使用する、薄いオプションのラッパーです。TLS インターフェイスの目的は、現在のソフトウェアの crypto ライブラリ、mbed TLS を、TLS プロトコルネゴシエーションと暗号化プリミティブの代替実装に簡単に置き換えることです。TLS インターフェイスは、SSL インターフェイスに変更を加えることなく交換することができます。FreeRTOS ソースコードリポジトリの「`iot_tls.h`」を参照してください。

SSL から crypto ライブラリのインターフェイスを直接選択できるよう、TLS インターフェイスはオプションです。このインターフェイスは、TLS およびネットワーク転送のフルスタックオフロード実装を含む MCU ソリューションには使用されません。

TLS インターフェイスの移植の詳細については、FreeRTOS 移植ガイドの TLS ライブラリの移植を参照してください。

Wi-Fi ライブラリ

> **Important**
>
> このライブラリは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

概要

FreeRTOS Wi-Fi ライブラリは、ポート別の Wi-Fi 実装を、Wi-Fi 機能を持つすべての FreeRTOS 認定ボードでのアプリケーション開発および移植を簡素化する共通の API に抽象化します。この共通の API を使用して、アプリケーションは、共通のインターフェイスを通じて低いレベルのワイヤレスのスタックと通信できます。

依存関係と要件

FreeRTOS Wi-Fi ライブラリには、FreeRTOS+TCP コアが必要です。

機能

Wi-Fi ライブラリには以下の機能があります。

- WEP、WPA、WPA2、WPA3 認証のサポート
- アクセスポイントのスキャン
- 電源管理
- ネットワークプロファイリング

Wi-Fi ライブラリの機能の詳細については、以下を参照してください。

Wi-Fi モード

Wi-Fi デバイスは、ステーションモード、アクセスポイントモード、P2P モードの 3 つのうちのいずれかのモードになります。Wi-Fi デバイスの現在のモードは、`WIFI_GetMode` を呼び出すことで取得できます。デバイスの Wi-Fi モードは、`WIFI_SetMode` を呼び出すことで設定できます。デバイスが既にネットワークに接続されている状態で `WIFI_SetMode` を呼び出してモードを切り替えると、接続は切断されます。

ステーションモード

デバイスをステーションモードに設定すると、ボードを既存のアクセスポイントに接続できます。

アクセスポイント (AP) モード

デバイスを AP モードに設定すると、そのデバイスを他のデバイスが接続できるアクセスポイントにすることができます。デバイスが AP モードになっている場合、FreeRTOS デバイスに別のデバイスを接続し、新しい Wi-Fi 認証情報を設定することができます。AP モードを設定するには、`WIFI_ConfigureAP` を呼び出します。デバイスを AP モードにするには、`WIFI_StartAP` を呼び出します。AP モードをオフにするには、`WIFI_StopAP` を呼び出します。

> **Note**
>
> FreeRTOS ライブラリは、AP モードでは Wi-Fi プロビジョニングを提供しません。AP モードを完全にサポートするには、DHCP および HTTP サーバー機能を含む追加の機能を提供する必要があります。

P2P モード

デバイスを P2P モードに設定すると、アクセスポイントなしで、複数のデバイスを直接相互に接続できるようにすることができます。

セキュリティ

Wi-Fi API は、WEP、WPA、WPA2、WPA3 セキュリティタイプをサポートしています。デバイスがステーションモードの場合は、`WIFI_ConnectAP` 関数を呼び出すときにネットワークセキュリティタイプを指定する必要があります。デバイスが AP モードの場合は、以下のサポートされているセキュリティタイプのいずれかを使用するようにデバイスを設定できます。

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

スキャンと接続

近くにあるアクセスポイントをスキャンするには、デバイスをステーションモードに設定し、WIFI_Scan 関数を呼び出します。スキャンで目的のネットワークが見つかった場合は、WIFI_ConnectAP を呼び出してそのネットワークの認証情報を指定することで、ネットワークに接続できます。ネットワークからの Wi-Fi デバイスの切断は、WIFI_Disconnect を呼び出すことで行えます。スキャンと接続の詳細については、「使用例」および「API リファレンス」を参照してください。

電源管理

さまざまな Wi-Fi デバイスやアプリケーションに応じて、要件が異なる電源を使用できます。Wi-Fi が不要な場合は、レイテンシーを短くするためにデバイスの電源をオンにするか、断続的に接続して低電力モードに切り替えることがあります。インターフェイス API は、常時オン、低電力、通常モードなどのさまざまな電力管理モードをサポートしています。WIFI_SetPMMode 関数を使用して、デバイスの電源モードを設定します。デバイスの現在の電源モードは、WIFI_GetPMMode 関数を呼び出すことで取得できます。

ネットワークプロファイル

Wi-Fi ライブラリを使用すると、ネットワークプロファイルをデバイスの不揮発性メモリに保存できます。これによりネットワーク設定が保存され、デバイスが Wi-Fi ネットワークに再接続したときにその設定が取得されるので、ネットワークに接続した後にデバイスを再プロビジョニングする必要がなくなります。WIFI_NetworkAdd はネットワークプロファイルを追加します。WIFI_NetworkGet は、ネットワークプロファイルを取得します。WIFI_NetworkDel はネットワークプロファイルを削除します。保存できるプロファイルの数は、プラットフォームによって異なります。

設定

Wi-Fi ライブラリを使用するには、設定ファイルで複数の ID を定義する必要があります。これらの ID の詳細については、API リファレンスを参照してください。

> **Note**
>
> ライブラリには、必要な設定ファイルは含まれていません。設定ファイルは作成する必要があります。設定ファイルを作成するときは、必ず、ボードに必要なボード固有の設定 ID を含める必要があります。

初期化

Wi-Fi ライブラリを使用する前には、FreeRTOS コンポーネントに加えて、一部のオンボード固有のコンポーネントを初期化する必要があります。初期化のテンプレートとして vendors/*vendor*/boards/*board*/aws_demos/application_code/main.c ファイルを使用し、以下の操作を行います。

1. ご自分のアプリケーションで Wi-Fi 接続を処理する場合は、main.c のサンプル Wi-Fi 接続ロジックを削除してください。以下の DEMO_RUNNER_RunDemos() 関数呼び出しを

   ```
   if( SYSTEM_Init() == pdPASS )
       {
       ...
           DEMO_RUNNER_RunDemos();
       ...
           }
   ```

   ご自分のアプリケーションの呼び出しに置き換えます。

   ```
   if( SYSTEM_Init() == pdPASS )
       {
       ...
           // This function should create any tasks
           // that your application requires to run.
           YOUR_APP_FUNCTION();
       ...
           }
   ```

2. WIFI_On() を呼び出して、Wi-Fi チップを初期化して電源を入れます。

   > **Note**
   > 一部のボードでは、追加のハードウェア初期化処理が必要になる場合があります。

3. 設定済みの WIFINetworkParams_t 構造を WIFI_ConnectAP() に渡して、利用可能な Wi-Fi ネットワークにボードを接続します。WIFINetworkParams_t 構造の詳細については、「使用例」および「API リファレンス」を参照してください。

API リファレンス

完全な API リファレンスについては、Wi-Fi API リファレンスを参照してください。

使用例

既知の AP への接続

```
#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
}
```

近くにある AP のスキャン

```
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);
}
```

移植

iot_wifi.c 実装は、iot_wifi.h で定義された関数を実装する必要があります。少なくとも、不可欠でない関数またはサポートされていない関数では、この実装は eWiFiNotSupported を返す必要があります。

Wi-Fi ライブラリを移植する方法については、FreeRTOS 移植ガイドの Wi-Fi ライブラリの移植を参照してください。

FreeRTOS デモ

FreeRTOS では、メイン FreeRTOS ディレクトリの demos フォルダにデモアプリケーションがいくつか含まれています。FreeRTOS で実行できるすべてのサンプルは、demos の下の common フォルダに表示されます。demos フォルダの下に、各 FreeRTOS 認定プラットフォーム用のフォルダもあります。

デモアプリケーションを試す前に、FreeRTOS の使用を開始する のチュートリアルを完了することをお勧めします。coreMQTT エージェントデモを設定して実行する方法について説明しています。

FreeRTOS デモを実行する

以下のトピックでは、FreeRTOS デモを設定して実行する方法について説明します。

- Bluetooth Low Energy デモアプリケーション
- Microchip Curiosity PIC32MZEF 用のデモブートローダー
- AWS IoT Device Defender デモ
- AWS IoT Greengrass V1 検出デモアプリケーション
- AWS IoT Greengrass V2
- coreHTTP のデモ

- AWS IoT ジョブライブラリのデモ
- coreMQTT デモ
- 無線通信経由更新デモアプリケーション
- セキュアソケットエコークライアントのデモ
- AWS IoT Device Shadow デモアプリケーション

DEMO_RUNNER_RunDemos 関数は *freertos*/demos/demo_runner/iot_demo_runner.c ファイルにあり、1 つのデモアプリケーションが動作する、デタッチされたスレッドを初期化します。デフォルトでは、DEMO_RUNNER_RunDemos が実行するのは coreMQTT エージェントデモの呼び出しと開始のみです。FreeRTOS をダウンロードしたときに選択した設定に応じて、また FreeRTOS をダウンロードした場所に応じて、ランナー関数の他の例がデフォルトで開始します。デモアプリケーションを有効にするには、*freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h ファイルを開き、実行するデモを定義します。

> **Note**
>
> サンプルのすべての組み合わせが使用できるわけではありません。組み合わせによっては、メモリの制約のために、選択したターゲット上でソフトウェアを実行できないことがあります。一度に 1 つのデモを実行することをお勧めします。

デモを設定する

このデモを迅速に開始できるように設定されています。お使いのプラットフォームで動作するバージョンを作成するために、プロジェクトの設定の一部を変更する必要があるかもしれません。設定ファイルは vendors/*vendor*/boards/*board*/aws_demos/config_files にあります。

Bluetooth Low Energy デモアプリケーション

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

概要

FreeRTOS Bluetooth Low Energy には 3 つのデモアプリケーションがあります。

- MQTT over Bluetooth Low Energy デモ

  このアプリケーションは、MQTT over Bluetooth Low Energy サービスの使用方法を示します。
- Wi-Fi プロビジョニング デモ

  このアプリケーションは、Bluetooth Low Energy Wi-Fi プロビジョニングサービスの使用方法を示します。
- 汎用属性サーバー デモ

  このアプリケーションは、FreeRTOS Bluetooth Low Energy ミドルウェア API を使用して単純な GATT サーバーを作成する方法を示します。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

前提条件

これらのデモを観るには、Bluetooth Low Energy 機能を備えたマイクロコントローラーが必要です。また、FreeRTOS Bluetooth デバイス用の iOS SDK または FreeRTOS Bluetooth デバイス用の Android SDK も必要です。

FreeRTOS Bluetooth Low Energy 用の AWS IoT と Amazon Cognito のセットアップ

MQTT AWS IoT 間でデバイスを に接続するには、 AWS IoT と Amazon Cognito を設定する必要があります。

をセットアップするには AWS IoT

1. https://aws.amazon.com/ で AWS アカウントを設定します。
2. AWS IoT コンソールを開き、ナビゲーションペインから [管理]、[モノ] の順に選択します。
3. [作成]、[単一のモノを作成する] の順に選択します。
4. デバイスの名前を入力し、[次へ] を選択します。

5. モバイルデバイスを使用して、マイクロコントローラーをクラウドに接続する場合は、[証明書なしでモノを作成] を選択します。Mobile SDK は Amazon Cognito を使用してデバイス認証を行っているため、Bluetooth Low Energy を使用するデモのためにデバイス証明書を作成する必要はありません。

   Wi-Fi を使用してマイクロコントローラーをクラウドディレクトリに接続する場合は、[証明書の作成]、[有効化] の順に選択し、モノの証明書、パブリックキー、プライベートキーをダウンロードします。
6. 登録したモノのリストから、先ほど作成したモノを選択し、モノのページから [操作] を選択します。REST API AWS IoT エンドポイントを書き留めます。

設定の詳細については、「 の開始方法 AWS IoT」を参照してください。

Amazon Cognito ユーザープールを作成するには

1. Amazon Cognito コンソールを開き、[Manage User Pools] (ユーザープールの管理) を選択します。
2. [ユーザープールを作成] を選択します。
3. ユーザープールに名前を付け、[デフォルトを確認する] を選択します。
4. ナビゲーションペインから、[アプリクライアント]、[アプリクライアントの追加] の順に選択します。
5. アプリクライアントの名前を入力し、[アプリクライアントの作成] を選択します。
6. ナビゲーションバーから、[Review]、[プールの作成] の順に選択します。

   ユーザープールの [全般設定] に表示されるプール ID を書き留めます。
7. ナビゲーションペインから、[アプリクライアント]、[詳細を表示] の順に選択します。アプリクライアント ID およびアプリクライアントシークレットを書き留めます。

Amazon Cognito アイデンティティプールを作成するには

1. Amazon Cognito コンソールを開き、[Manage Identity Pools] (ID プールの管理) を選択します。
2. ID プールの名前を入力します。
3. [認証プロバイダー] を展開して [Cognito] を選択し、ユーザープール ID およびアプリクライアント ID を入力します。
4. [プールの作成] を選択します。

5. [詳細を表示] を展開し、2 種類の IAM ロール名を書き留めます。[Allow] (許可) を選択し、認証済み ID および未認証 ID の Amazon Cognito へのアクセスを認可する IAM ロールを作成します。
6. [Edit identity pool] (ID プールの編集) をクリックします。ID プールの ID を書き留めます。形式は `us-west-2:12345678-1234-1234-1234-123456789012` です。

Amazon Cognito のセットアップの詳細については、Amazon Cognito の使用開始方法を参照してください。

IAM ポリシーを作成して、認証済み ID にアタッチするには

1. IAM コンソールを開き、ナビゲーションペインで [Roles] (ロール) を選択します。
2. 認証された ID のロールを検索して選択し、[Attach policies (ポリシーのアタッチ)]、[Add inline policy (インラインポリシーの追加)] の順に選択します。
3. [JSON] タブを選択し、次の JSON を貼り付けます。

```
{
   "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. [ポリシーの確認] を選択してポリシーの名前を入力し、[ポリシーの作成] を選択します。

AWS IoT と Amazon Cognito の情報を手元に保管します。 AWS クラウドでモバイルアプリケーションを認証するには、エンドポイントと IDs が必要です。

Bluetooth Low Energy 用の FreeRTOS 環境をセットアップする

ご利用の環境をセットアップするには、Bluetooth Low Energy ライブラリ を使用してマイクロコントローラーに FreeRTOS をダウンロードし、FreeRTOS Bluetooth デバイス用 Mobile SDK をご利用のモバイルデバイスにダウンロードして設定する必要があります。

FreeRTOS Bluetooth Low Energy でマイクロコントローラーの環境をセットアップするには

1. FreeRTOS を GitHub からダウンロードするか、クローンします。手順については、README.md ファイルを参照してください。
2. マイクロコントローラーに FreeRTOS をセットアップします。

   FreeRTOS の資格を満たしたマイクロコントローラーで FreeRTOS の使用を開始する方法については、ご利用のボードの FreeRTOS の使用開始ガイドを参照してください。

   > **Note**
   >
   > デモは、FreeRTOS と移植された FreeRTOS Bluetooth Low Energy ライブラリを搭載した Bluetooth Low Energy 対応マイクロコントローラーで実行できます。現在、FreeRTOS MQTT over Bluetooth Low Energy デモプロジェクトは、以下の Bluetooth Low Energy 対応デバイスに完全に移植されます。
   >
   > - Espressif ESP32-DevKitC and the ESP-WROVER-KIT
   > - Nordic nRF52840-DK

共通コンポーネント

FreeRTOS デモアプリケーションには、2 つの共通のコンポーネントが含まれています。

- Network Manager
- Bluetooth Low Energy Mobile SDK デモアプリケーション

Network Manager

Network Manager は、マイクロコントローラーのネットワーク接続を管理します。demos/network_manager/aws_iot_network_manager.c の FreeRTOS ディレクトリにあります。Network Manager が Wi-Fi と Bluetooth Low Energy の両方に対応している場合、デフォルトでは Bluetooth Low Energy を使ってデモが開始されます。Bluetooth Low Energy 接続が中断され、ボードが Wi-Fi に対応している場合、Network Manager は使用可能な Wi-Fi 接続に切り替えてネットワークからの切断を回避します。

Network Manager でネットワーク接続タイプを有効にするには、vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_iot_network_config.h で configENABLED_NETWORKS パラメータにネットワーク接続タイプを追加します (*vendor* はベンダーの名前で、*board* はデモを実行するために使用中のボードの名前です)。

たとえば、Bluetooth Low Energy と Wi-Fi の両方が有効な場合、aws_iot_network_config.h にある #define configENABLED_NETWORKS で始まる行は次のようになります。

```
#define  configENABLED_NETWORKS  ( AWSIOT_NETWORK_TYPE_BLE | AWSIOT_NETWORK_TYPE_WIFI )
```

現在サポートされているネットワーク接続タイプのリストを取得するには、aws_iot_network.h の #define AWSIOT_NETWORK_TYPE から始まる行を参照してください。

FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション

FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションは、GitHub から取得でき、FreeRTOS Bluetooth デバイス用 Android SDK は amazon-freertos-ble-android-sdk/app に、FreeRTOS Bluetooth デバイス用 iOS SDK は amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo にあります。この例では、iOS バージョンのデモモバイルアプリケーションのスクリーンショットを使用します。

> **Note**
>
> iOS デバイスを使用している場合は、デモ用モバイルアプリケーションを構築するために Xcode が必要です。Android デバイスを使用している場合、Android Studio を使用してデモ用モバイルアプリケーションを構築できます。

iOS SDK デモアプリケーションを設定するには

設定変数を定義する場合は、設定ファイルで指定されているプレースホルダー値の形式を使用します。

1. FreeRTOS Bluetooth デバイス用の iOS SDK がインストールされていることを確認します。
2. `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/` から次のコマンドを発行します。

   ```
   $ pod install
   ```

3. Xcode で `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo.xcworkspace` プロジェクトを開き、署名している開発者アカウントを自分のアカウントに変更します。
4. リージョンに AWS IoT ポリシーを作成します (まだ作成していない場合) 。

   > **Note**
   >
   > このポリシーは、Amazon Cognito 認証済み ID 用に作成された IAM ポリシーとは異なります。

   a. AWS IoT コンソールを開きます。

   b. ナビゲーションペインで、[Secure] (保護) を選択し、[Policies] (ポリシー) を選択してから [Create] (作成) を選択します。ポリシーを識別するための名前を入力します。[Add statements] (ステートメントを追加) セクションで、[Advanced mode] (アドバンストモード) を選択します。次の JSON をポリシーエディタウィンドウにコピーして貼り付けます。*aws-region* と *aws-account* を自分の AWS リージョンとアカウント ID に置き換えます。

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

```
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:region:account-id:*"
        },
        {
             "Effect": "Allow",
             "Action": "iot:Subscribe",
             "Resource": "arn:aws:iot:region:account-id:*"
        },
        {
             "Effect": "Allow",
             "Action": "iot:Receive",
             "Resource": "arn:aws:iot:region:account-id:*"
        }
    ]
}
```

c. [Create] (作成) を選択します。

5. `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo/Amazon/AmazonConstants.swift` を開き、次の変数を再定義します。

   - `region`: AWS リージョン。
   - `iotPolicyName`: AWS IoT ポリシー名。
   - `mqttCustomTopic`: 発行する MQTT トピック。

6. `amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo/Support/awsconfiguration.json` を開きます。

   `CognitoIdentity` で、次の変数を再定義します。

   - `PoolId`: Amazon Cognito ID プール ID。
   - `Region`: AWS リージョン。

   `CognitoUserPool` で、次の変数を再定義します。

   - `PoolId`: Amazon Cognito ユーザープール ID。
   - `AppClientId`: アプリクライアント ID。
   - `AppClientSecret`: アプリクライアントシークレット。
   - `Region`: AWS リージョン。

Android SDK デモアプリケーションを設定するには

設定変数を定義する場合は、設定ファイルで指定されているプレースホルダー値の形式を使用します。

1. FreeRTOS Bluetooth デバイス用の Android SDK がインストールされていることを確認します。
2. リージョンに AWS IoT ポリシーを作成します (まだ作成していない場合) 。

> **Note**
>
> このポリシーは、Amazon Cognito 認証済み ID 用に作成された IAM ポリシーとは異なります。

   a. AWS IoT コンソールを開きます。
   b. ナビゲーションペインで、[Secure] (保護) を選択し、[Policies] (ポリシー) を選択してから [Create] (作成) を選択します。ポリシーを識別するための名前を入力します。[Add statements] (ステートメントを追加) セクションで、[Advanced mode] (アドバンストモード) を選択します。次の JSON をポリシーエディタウィンドウにコピーして貼り付けます。*aws-region* と *aws-account* を自分の AWS リージョンとアカウント ID に置き換えます。

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

```
        {
             "Effect": "Allow",
             "Action": "iot:Receive",
             "Resource": "arn:aws:iot:region:account-id:*"
        }
    ]
}
```

c. [Create] (作成) を選択します。

3. https://github.com/aws/amazon-freertos-ble-android-sdk/blob/master/app/src/main/java/software/amazon/freertos/demo/DemoConstants.java を開き、以下の変数を再定義します。

   - `AWS_IOT_POLICY_NAME`: AWS IoT ポリシー名。
   - `AWS_IOT_REGION`: AWS リージョン。

4. https://github.com/aws/amazon-freertos-ble-android-sdk/blob/master/app/src/main/res/raw/awsconfiguration.json を開きます。

   `CognitoIdentity` で、次の変数を再定義します。

   - `PoolId`: Amazon Cognito ID プール ID。
   - `Region`: AWS リージョン。

   `CognitoUserPool` で、次の変数を再定義します。

   - `PoolId`: Amazon Cognito ユーザープール ID。
   - `AppClientId`: アプリクライアント ID。
   - `AppClientSecret`: アプリクライアントシークレット。
   - `Region`: AWS リージョン。

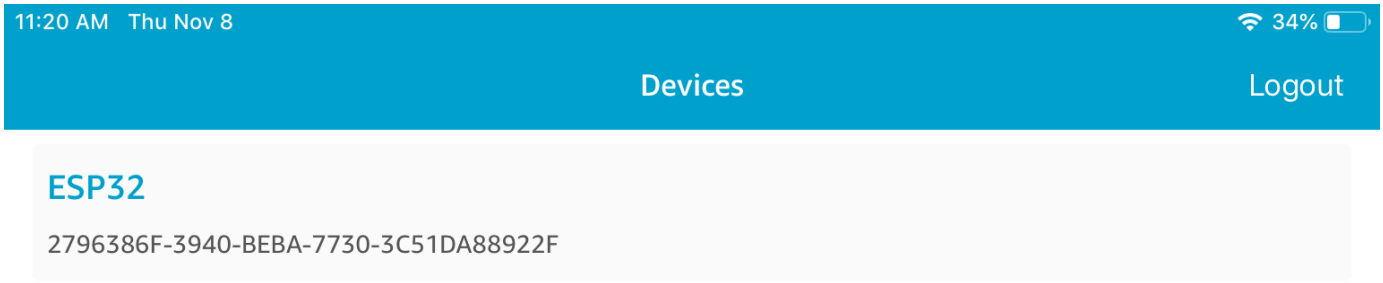
Bluetooth Low Energy 経由でマイクロコントローラーとの安全な接続を検出して確立するには

1. マイクロコントローラとモバイルデバイスを安全にペアリングするには (ステップ 6)、入力と出力の両方の機能 (TeraTerm など) を備えたシリアルターミナルエミュレータが必要です。ターミナルエミュレーターをインストールする の手順に従って、ターミナルをシリアル接続でボードに接続するよう設定します。
2. マイクロコントローラーで Bluetooth Low Energy デモプロジェクトを実行します。
3. モバイルデバイスで Bluetooth Low Energy Mobile SDK デモアプリケーションを実行します。

コマンドラインから Android SDK のデモアプリケーションを開始するには、次のコマンドを実行します。

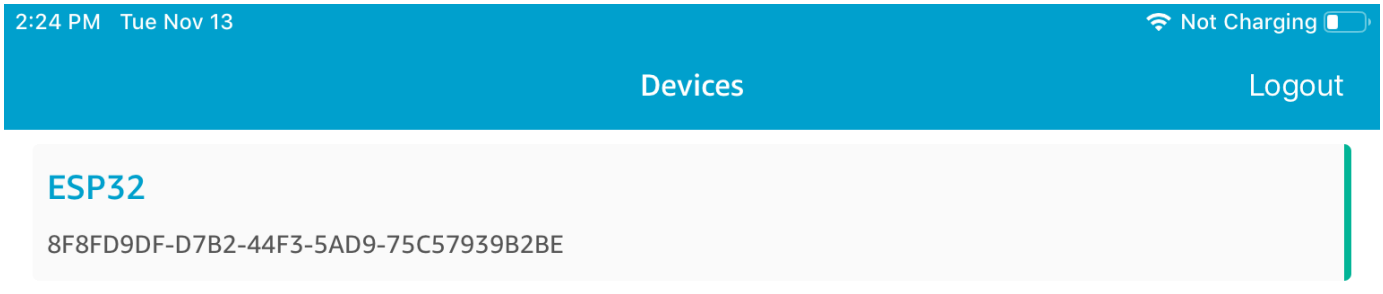
```
$ ./gradlew installDebug
```

4. Bluetooth Low Energy Mobile SDK デモアプリの [デバイス] にマイクロコントローラーが表示されていることを確認します。

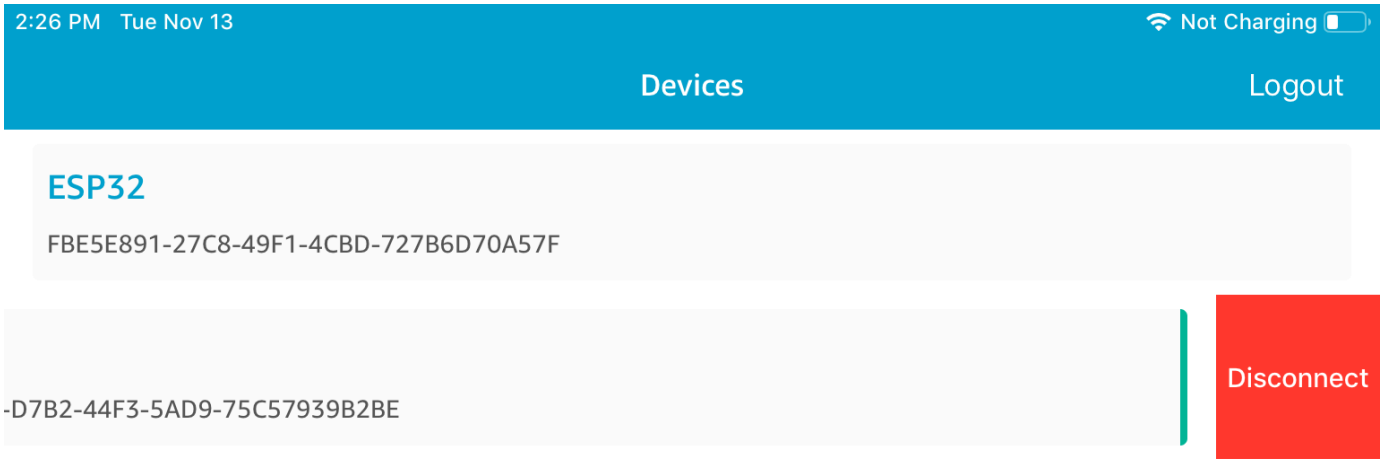


> **Note**
>
> このリストには、 FreeRTOS を搭載したすべてのデバイスと、範囲内にあるデバイス情報サービス (*freertos*/.../device_information) が表示されます。

5. デバイスのリストからマイクロコントローラーを選択します。アプリケーションがボードとの接続を確立し、接続されたデバイスの横に緑色の線が表示されます。

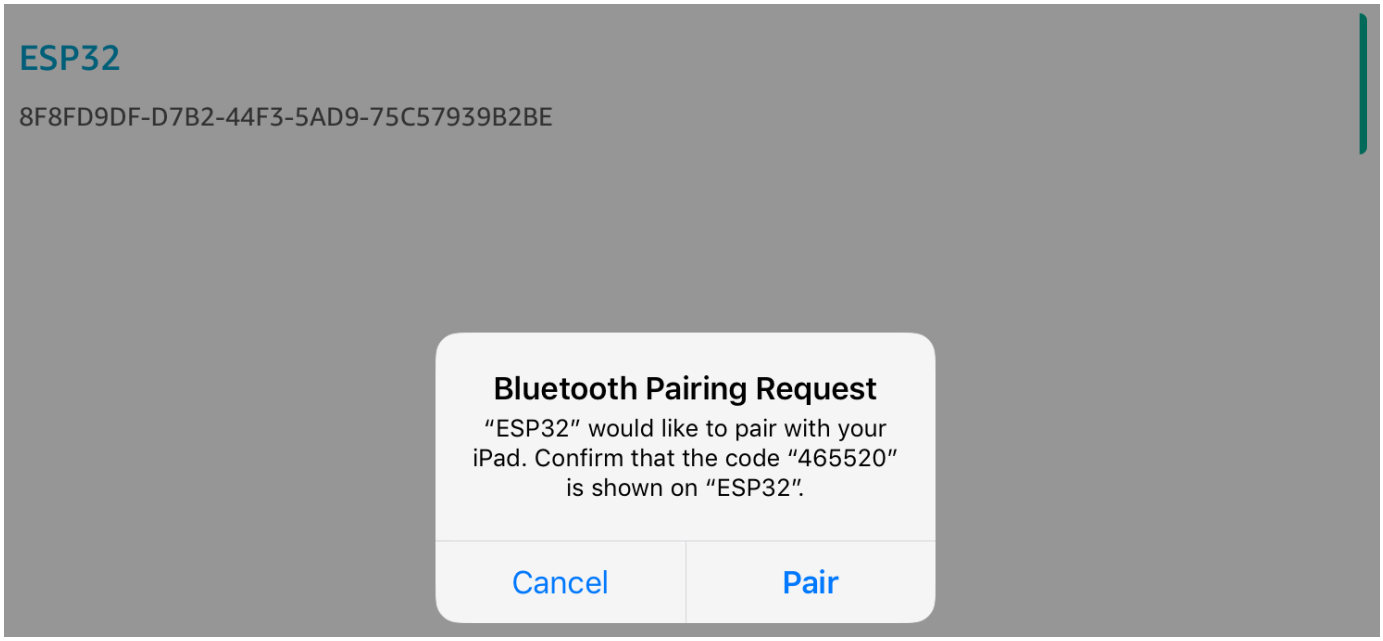


マイクロコントローラーとの接続を切断するには、その線を左へドラッグします。



6. プロンプトが表示されたら、マイクロコントローラとモバイルデバイスをペアリングします。

```
24 261107 [Btc_task]  Disconnected from BLE device. Stopping the counter update
25 261108 [Btc_task] Disconnect received for MQTT service instance 0
26 261108 [Btc_task] BLE disconnected with remote device, start advertisement
27 261108 [Btc_task] Started advertisement. Listening for a BLE Connection.
28 261289 [Btc_task] BLE Connected to remote device, connId = 0
E (2614110) BT_GATT: gatts_write_attr_perm_check - GATT_INSUF_AUTHENTICATION: MITM required
E (2614420) BT_SMP: Value for numeric comparison = 465520
29 261412 [uTask] Numeric comparison:465520
30 261412 [uTask] Press 'y' to confirm
```



数値比較用のコードが両方のデバイスで同じである場合は、デバイスをペアリングします。

> **Note**
>
> Bluetooth Low Energy Mobile SDK デモアプリケーションは、ユーザー認証に Amazon Cognito を使用します。Amazon Cognito ユーザーおよび ID プールが設定されていることと、認証済み ID に IAM ポリシーがアタッチされていることを確認します。

MQTT over Bluetooth Low Energy

MQTT over Bluetooth Low Energy デモでは、マイクロコントローラーは MQTT プロキシを介してメッセージを AWS クラウドに発行します。

デモ MQTT トピックをサブスクライブするには

1. AWS IoT コンソールにサインインします。
2. ナビゲーションペインで、[テスト] を選択し、[MQTT テストクライアント] を選択して MQTT クライアントを開きます。
3. [Subscription topic] (トピックのサブスクリプション) で ***thing-name*/example/topic1** と入力し、[Subscribe to topic] (トピックへのサブスクライブ) を選択します。

Bluetooth Low Energy を使用してマイクロコントローラーとモバイルデバイスのペアリングを行う場合、MQTT メッセージは、モバイルデバイスの Bluetooth Low Energy Mobile SDK デモアプリケーションを通じてルーティングされます。

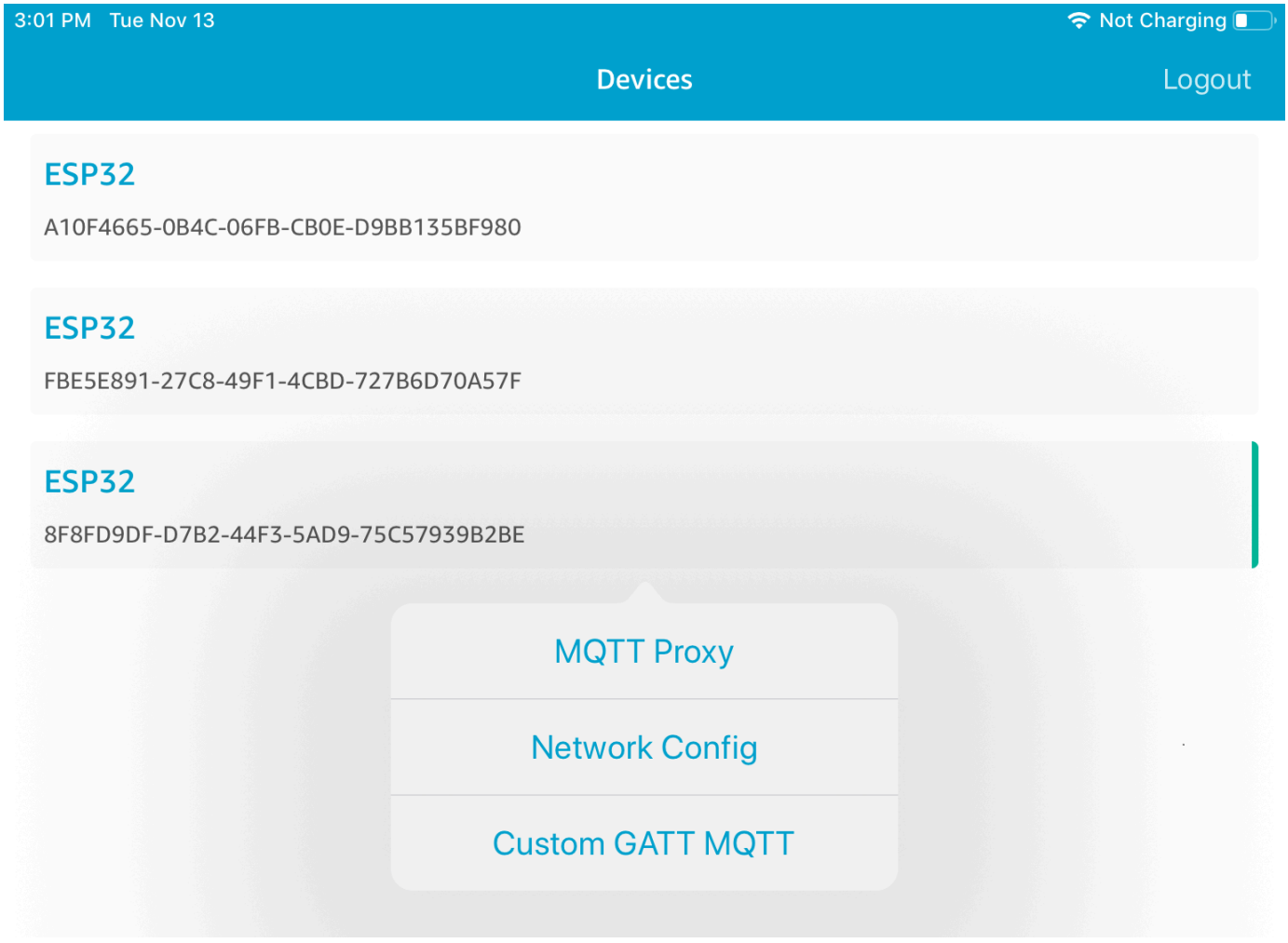
Bluetooth Low Energy 経由でデモを有効にするには

1. `vendors/vendor/boards/board/aws_demos/config_files/aws_demo_config.h` を開き、`CONFIG_MQTT_BLE_TRANSPORT_DEMO_ENABLED` を定義します。
2. を開き`demos/include/aws_clientcredential.h`、 AWS IoT ブローカーエンドポイント`clientcredential`で `MQTT_BROKER_ENDPOINT` を設定します。BLE マイクロコントローラーデバイスのモノの名前で `clientcredential`を設定します。 `IOT_THING_NAME` AWS IoT ブローカーエンドポイントは、左側のナビゲーションペインで 設定 を選択して AWS IoT コンソールから取得するか、 コマンド を実行して CLI から取得できます`aws iot describe-endpoint --endpoint-type=iot:Data-ATS`。

> **Note**
>
> AWS IoT ブローカーエンドポイントとモノの名前はどちらも、シークレット ID とユーザープールが設定されているのと同じリージョンに存在する必要があります。

デモを実行するには

1. マイクロコントローラーでデモプロジェクトを構築し、実行します。
2. ボードとモバイルデバイスが、FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション を使ってペアリングされていることを確認します。
3. デモモバイルアプリの [デバイス] リストからマイクロコントローラーを選択し、[MQTT プロキシ] を選択して MQTT プロキシ設定を開きます。



4. MQTT プロキシを有効にすると、*thing-name*/example/topic1 トピックに MQTT メッセージが表示され、データが UART ターミナルに出力されます。

Wi-Fi プロビジョニング

Wi-Fi プロビジョニングは、Bluetooth Low Energy 経由でモバイルデバイスからマイクロコントローラーに Wi-Fi ネットワーク認証情報を安全に送信できる FreeRTOS Bluetooth Low Energy サービスです。Wi-Fi プロビジョニングサービスのソースコードは、*freertos*/.../wifi_provisioning にあります。

> **Note**
>
> Wi-Fi プロビジョニングデモは、現在、Espressif ESP32-DevKitC でサポートされています。

デモを有効化するには

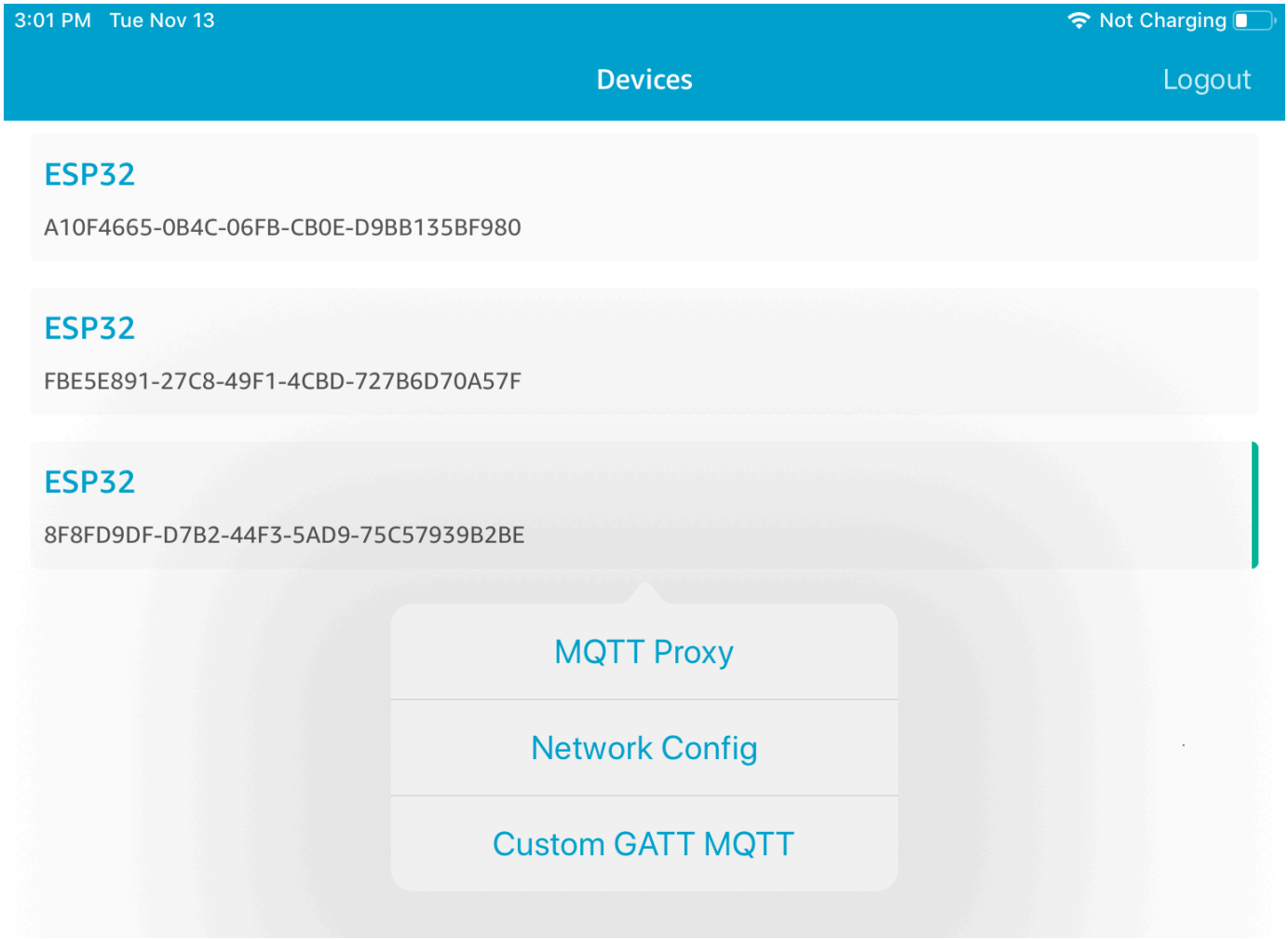
1. Wi-Fi プロビジョニングサービスを有効にします。vendors/*vendor*/boards/*board*/aws_demos/config_files/iot_ble_config.h を開き、#define IOT_BLE_ENABLE_WIFI_PROVISIONING を 1 に設定します (*vendor* にはベンダーの名前、*board* にはデモの実行に使用しているボードの名前が入ります)。

   > **Note**
   >
   > Wi-Fi プロビジョニングサービスはデフォルトでは無効になっています。

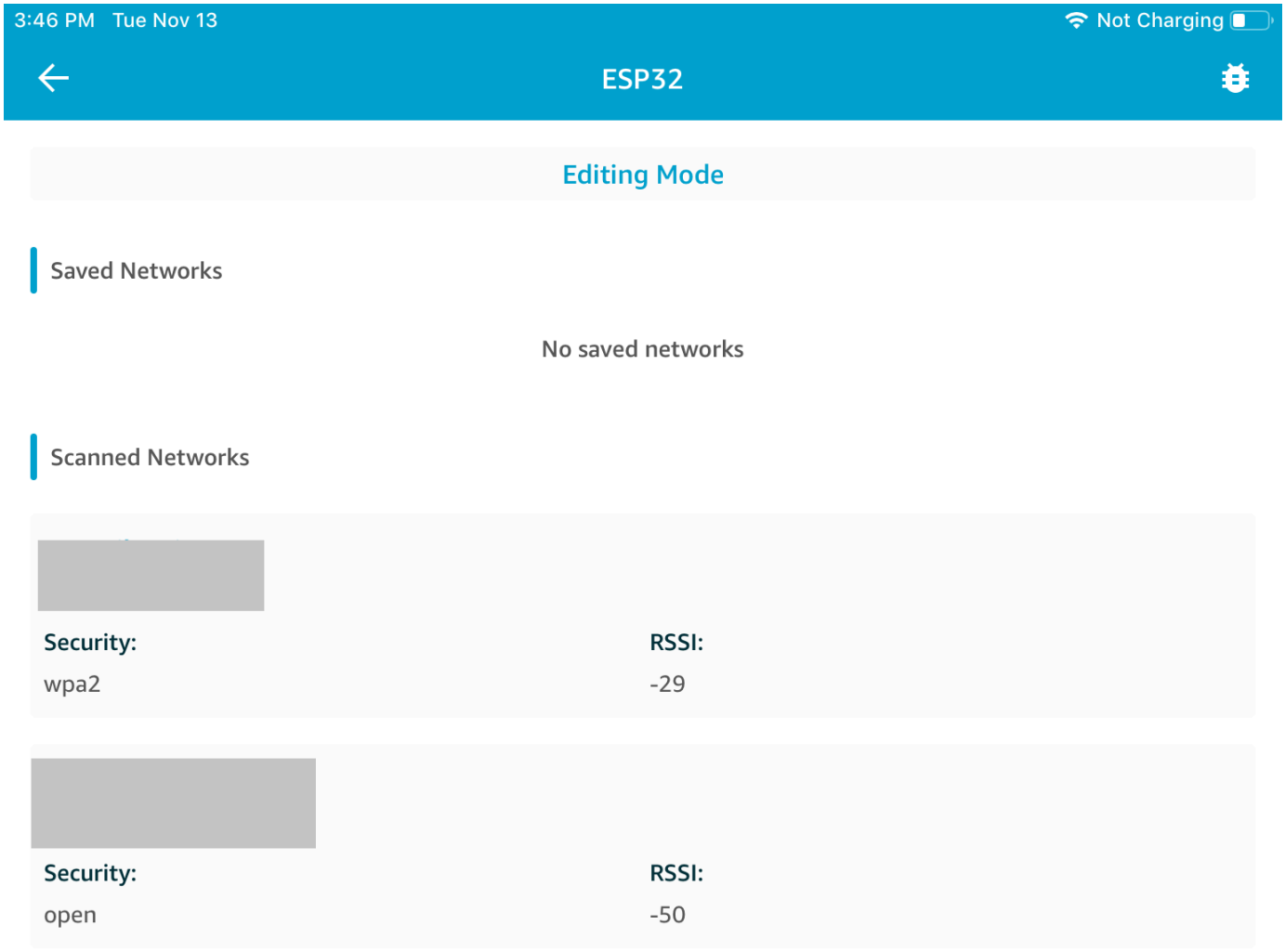
2. Bluetooth Low Energy と Wi-Fi の両方を有効にするよう Network Manager を設定します。

デモを実行するには

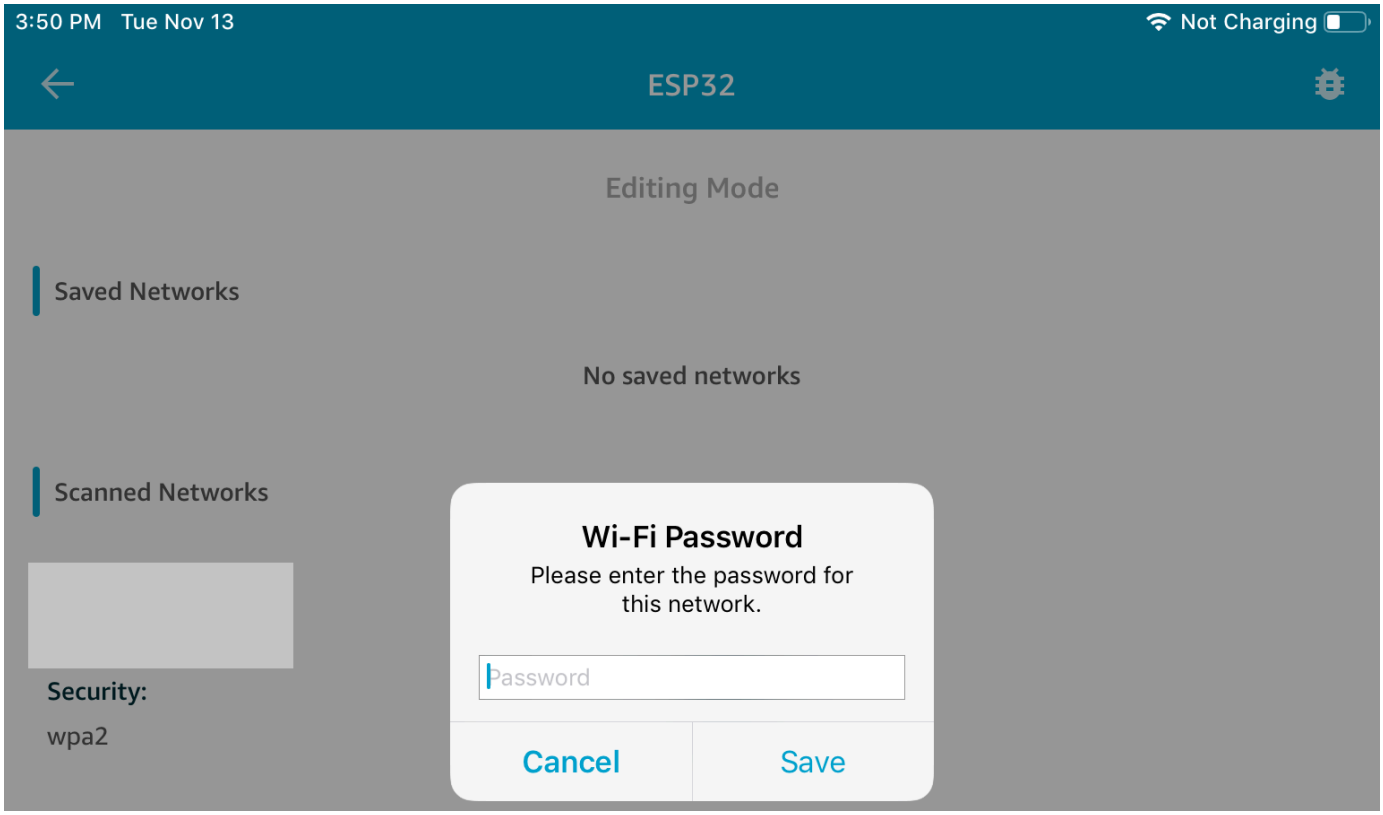
1. マイクロコントローラーでデモプロジェクトを構築し、実行します。
2. マイクロコントローラーとモバイルデバイスが、FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション を使ってペアリングされていることを確認します。
3. デモモバイルアプリの [Devices (デバイス)] リストからマイクロコントローラーを選択し、次に [Network Config (ネットワーク構成)] を選択してネットワーク構成設定を開きます。



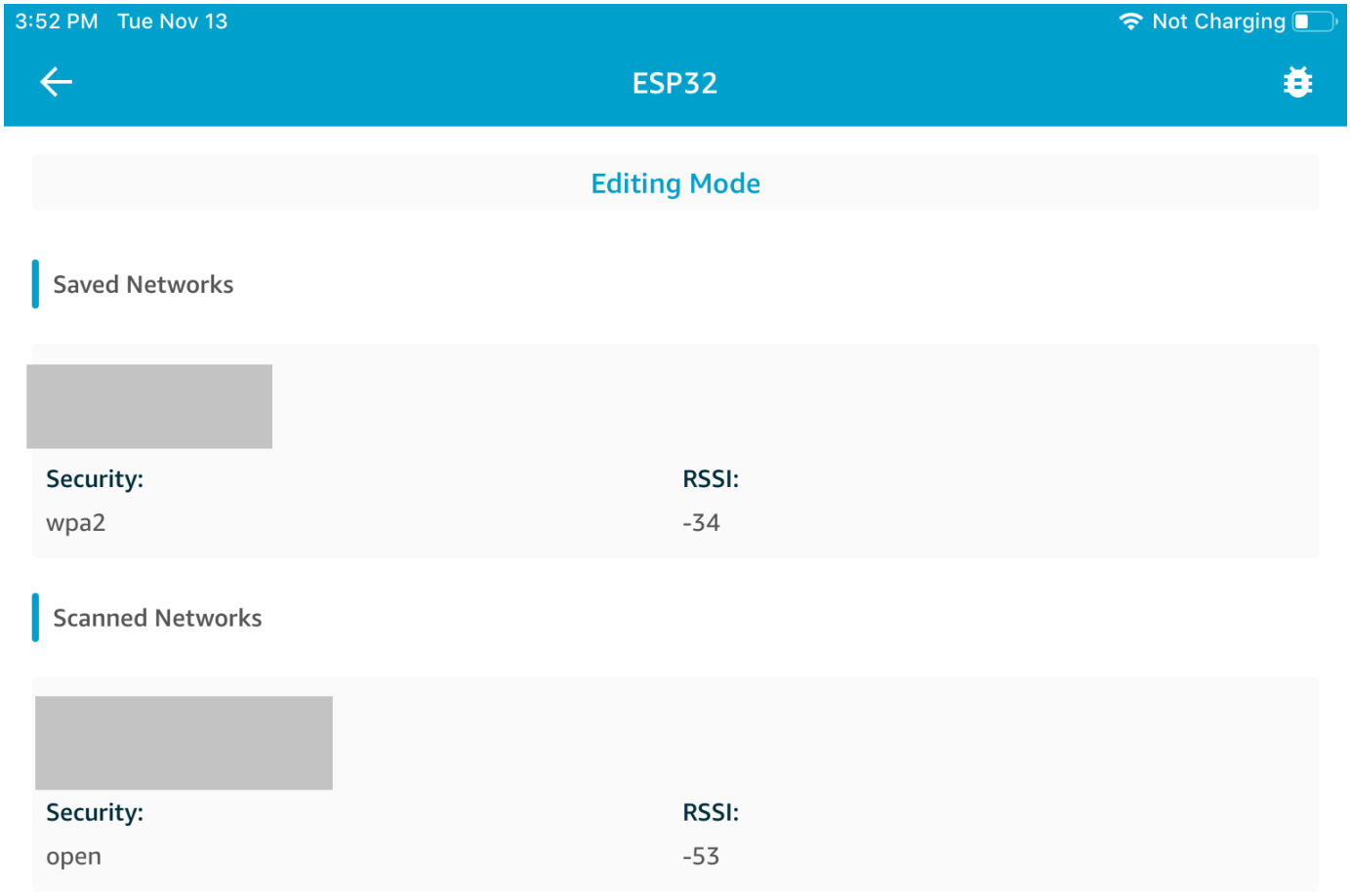
4. ボードの [Network Config (ネットワーク構成)] を選択した後、マイクロコントローラーが近隣にあるネットワークのリストをモバイルデバイスに送信します。[Scanned Networks (スキャンされたネットワーク)] のリストに利用可能な Wi-Fi ネットワークが表示されます。



[Scanned Networks (スキャンされたネットワーク)] のリストからネットワークを選択し、必要に応じて SSID とパスワードを入力します。

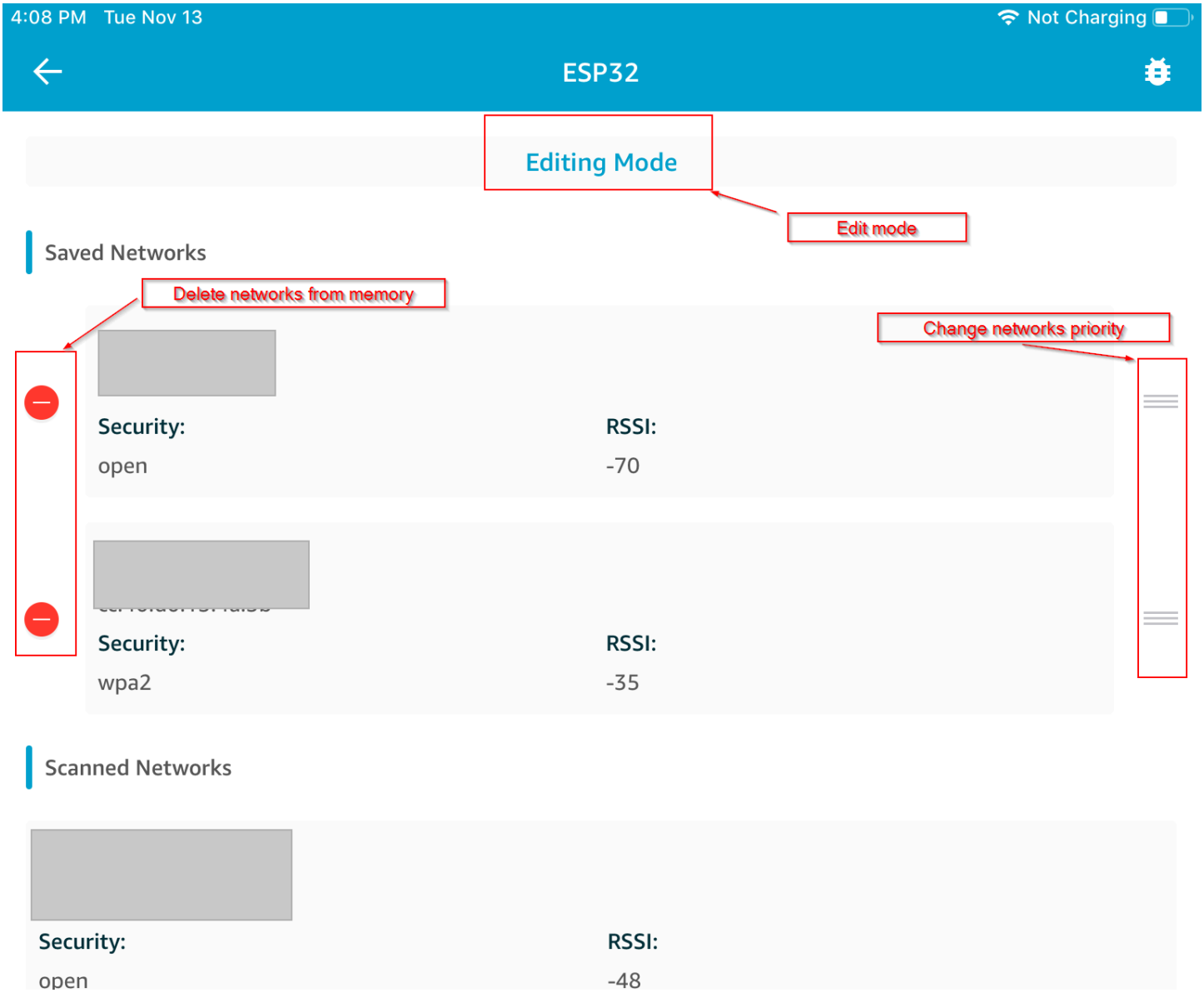


マイクロコントローラーがネットワークに接続し、ネットワークを保存します。ネットワークが [Saved Networks (保存されたネットワーク)] に表示されます。



デモモバイルアプリには複数のネットワークを保存できます。アプリケーションとデモを再起動すると、[Saved Networks (保存されたネットワーク)] の上から順に、最初に利用可能なネットワークに接続します。

ネットワークの優先順位を変更したり、ネットワークを削除したりするには、[Network Configuration (ネットワーク構成)] ページで [Editing Mode (編集モード)] を選択します。ネットワークの優先順位を変更するには、該当するネットワークの右側を選択し、上下にドラッグします。ネットワークを削除するには、該当するネットワークの左側にある赤色のボタンを選択します。



汎用属性サーバー

この例では、マイクロコントローラーにあるデモの Generic Attributes (GATT) サーバーアプリケーションが単純なカウンター値をFreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーションに送信します。

Bluetooth Low Energy Mobile SDK を使用すると、モバイルデバイス用に独自の GATT クライアントを作成することができます。このクライアントは、マイクロコントローラー上の GATT サーバーに接続し、デモモバイルアプリケーションと並行して実行されます。

デモを有効化するには

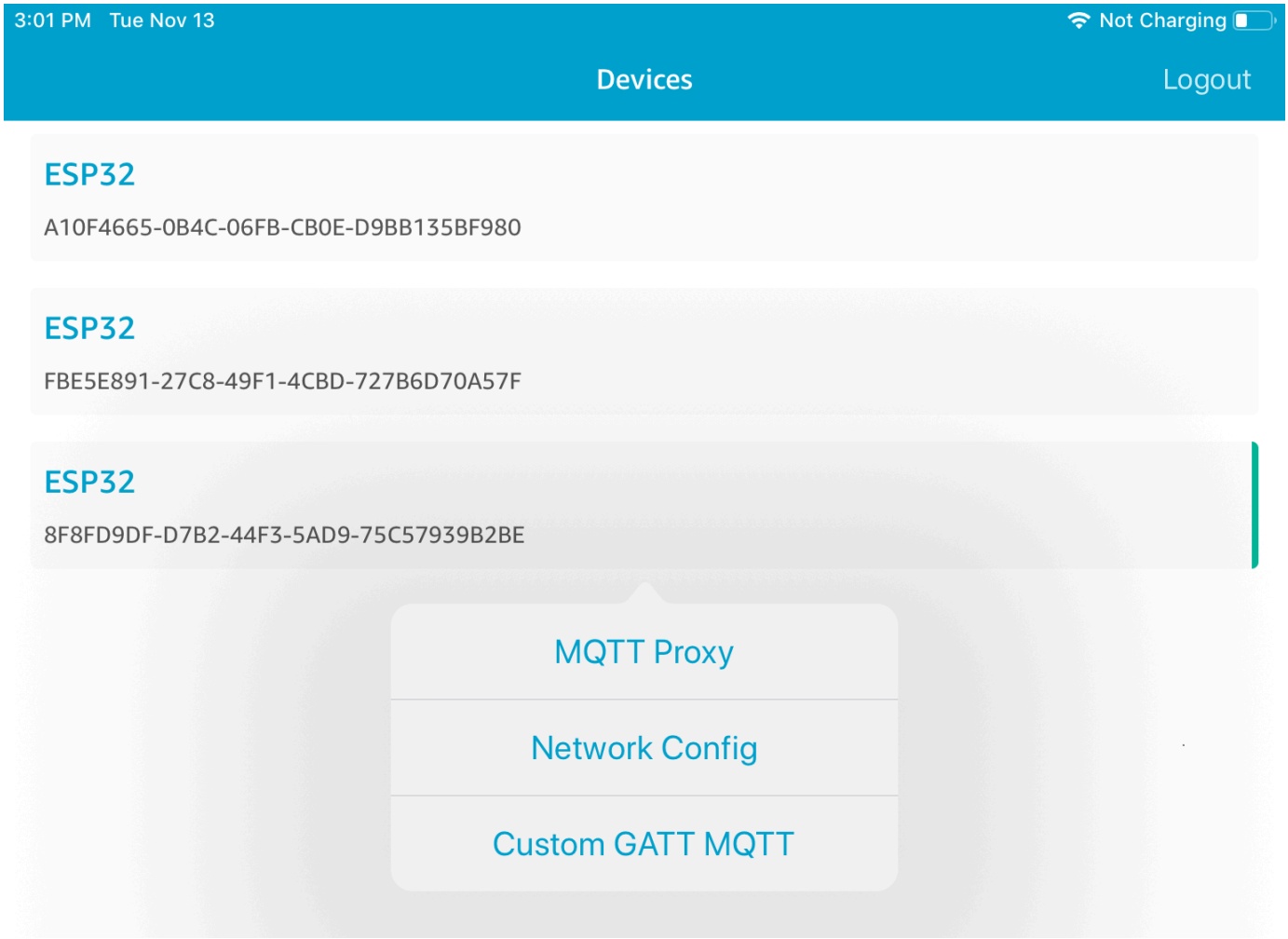
1. Bluetooth Low Energy GATT デモを有効にします。vendors/*vendor*/boards/*board*/aws_demos/config_files/iot_ble_config.h (*vendor* はベンダーの名前で、*board* はデモの実行に使用中のボードの名前です) で、define ステートメントのリストに #define IOT_BLE_ADD_CUSTOM_SERVICES (1) を追加します。

   > **Note**
   >
   > デフォルトでは、Bluetooth Low Energy GATT デモは無効になっています。

2. *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして CONFIG_BLE_GATT_SERVER_DEMO_ENABLED を定義します。

デモを実行するには

1. マイクロコントローラーでデモプロジェクトを構築し、実行します。
2. ボードとモバイルデバイスが、FreeRTOS Bluetooth Low Energy Mobile SDK デモアプリケーション を使ってペアリングされていることを確認します。
3. アプリの [デバイス] リストからボードを選択し、[MQTT プロキシ] を選択して MQTT プロキシオプションを開きます。



4. [Devices (デバイス)] リストに戻ってボードを選択し、次に [Custom GATT MQTT (カスタム GATT MQTT)] を選択してカスタムの GATT サービスオプションを開きます。
5. [Start Counter (カウンターの開始)] を選択して ***your-thing-name*/example/topic** MQTT トピックへのデータの発行を開始します。

   MQTT プロキシを有効にした後、*your-thing-name*/example/topic トピックに Hello World と増分カウンターメッセージが表示されます。

Microchip Curiosity PIC32MZEF 用のデモブートローダー

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-

FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

> **Note**
>
> Microchip との合意に基づき、Curiosity PIC32MZEF (DM320104) は FreeRTOS Reference Integration リポジトリのメインブランチから削除されるため、新しいリリースには搭載されません。Microchip は PIC32MZEF (DM320104) が新しい設計に推奨されなくなったことを公式に発表しました。PIC32MZEF プロジェクトとソースコードには、以前のリリースタグから引き続きアクセスできます。Microchip では、新しい設計に Curiosity PIC32MZ-EF-2.0 Development board (DM320209) を使用するよう推奨しています。Pic32mzv1 プラットフォームは、FreeRTOS リファレンス統合リポジトリの v202012.00 に引き続き用意されています。ただし、プラットフォームは FreeRTOS リファレンスの v202107.00 によってサポートされなくなりました。

このデモブートローダーは、ファームウェアバージョンチェック、暗号署名の検証、およびアプリケーションの自己テストを実装します。これらの機能は、FreeRTOS の無線通信経由 (OTA) によるファームウェアの更新をサポートします。

ファームウェア検証は、無線で受信した新しいファームウェアの信頼性と誠実性を検証することが含まれます。ブートローダーは、起動する前にアプリケーションの暗号署名を検証します。このデモでは、SHA-256 と楕円曲線デジタル署名アルゴリズム (ECDSA) を使用しています。提供されているユーティリティを使用して、デバイス上でフラッシュできる署名付きアプリケーションを生成することができます。

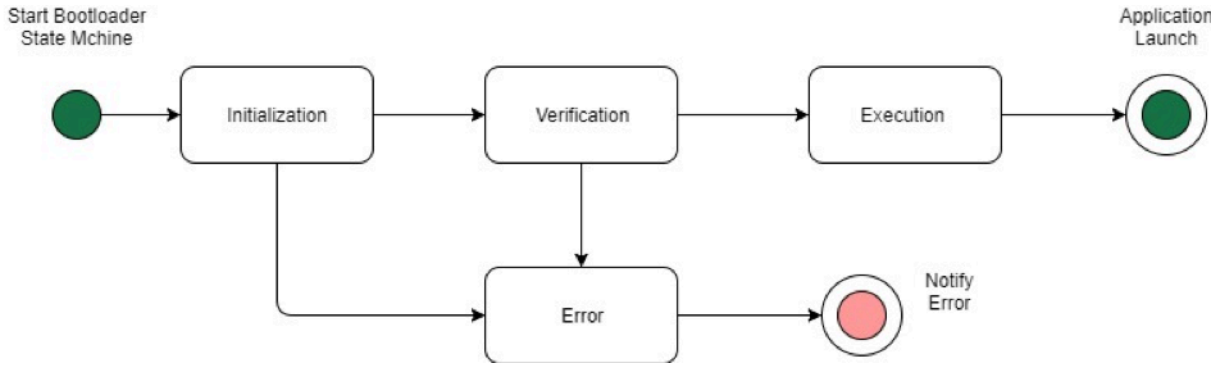
ブートローダーは、OTA に必要な次の機能をサポートしています。

- デバイス上でアプリケーションイメージを維持し、それらを切り替えます。
- 受信した OTA イメージの自己テスト実行を許可し、失敗時にロールバックします。
- OTA 更新イメージの署名とバージョンをチェックします。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

ブートローダーのステート

ブートローダーのプロセスは以下のステートマシンで示されます。



次の表はブートローダーのステートを説明しています。

ブートローダーのステート	説明
初期化	ブートローダーが初期化状態になっています。
検証	ブートローダーはデバイスに存在するイメージを検証しています。
イメージ実行	ブートローダーが選択したイメージを起動しています。
デフォルト実行	ブートローダーがデフォルトイメージを起動しています。
エラー	ブートローダーがエラー状態になっています。

上の図では、`Execute Image` および `Execute Default` の両方が `Execution` ステートとして表示されています。

ブートローダーの実行ステート

> ブートローダーは、`Execution` 状態にあり、選択した検証済みイメージを起動する準備ができました。起動されるイメージが上位バンクにある場合、アプリケーションは常に下位バンク用に構築されているため、イメージを実行する前にバンクがスワップされます。

ブートローダーのデフォルト実行ステート

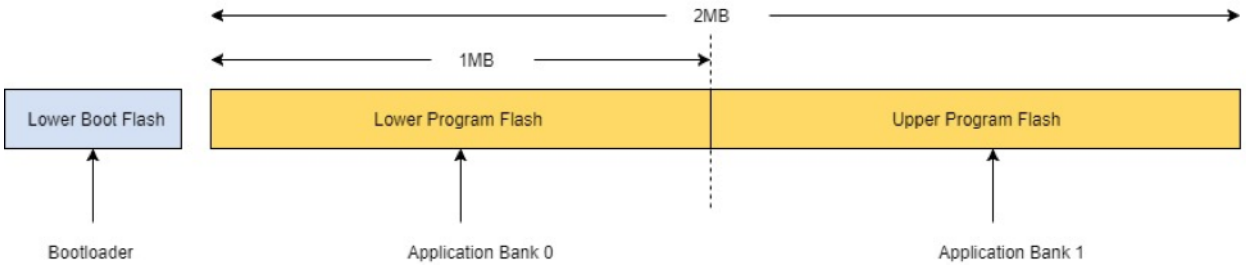
デフォルトイメージを起動する設定オプションが有効になっている場合、ブートローダーはデフォルトの実行アドレスからアプリケーションを起動します。デバッグ中を除いて、このオプションを無効にする必要があります。

ブートローダーのエラー状態

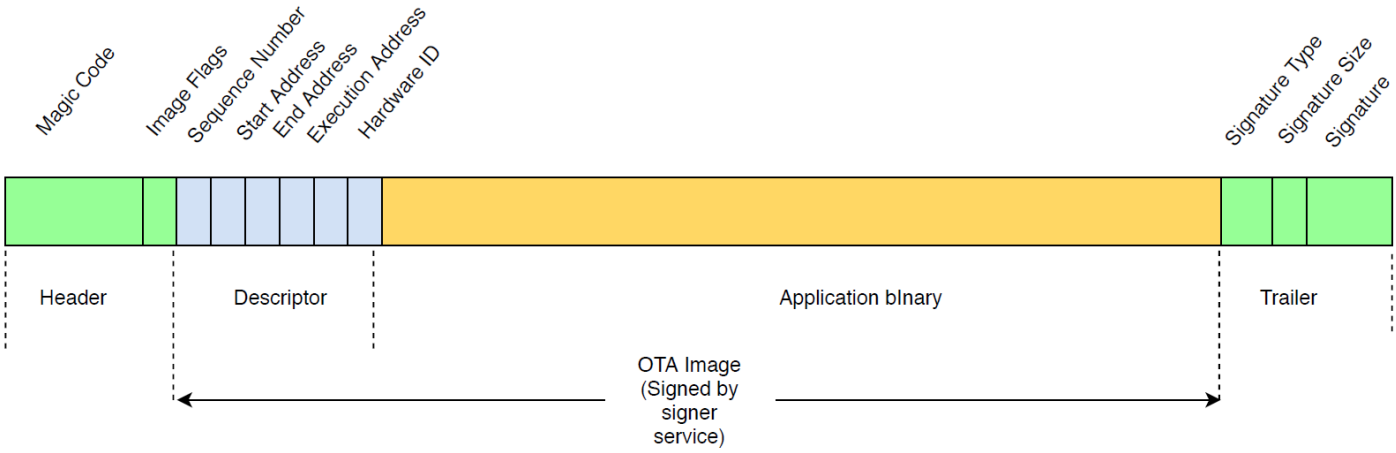
ブートローダーがエラー状態で、デバイスに有効なイメージがありません。ブートローダーはユーザーに通知する必要があります。デフォルトの実装では、ログメッセージがコンソールに送信され、ボード上の LED が永続的にすばやく点滅します。

フラッシュデバイス

Microchip Curiosity PIC32MZEF プラットフォームでは、2 メガバイト (MB) の内部プログラムフラッシュが 2 つのバンクに分割されています。これらの 2 つのバンクとライブアップデート間のメモリマップスワッピングをサポートしています。デモブートローダーは、別個の下部ブートフラッシュ領域にプログラムされています。



アプリケーションイメージ構造



この図は、デバイスの各バンクに保存されているアプリケーションイメージのプライマリコンポーネントを示しています。

コンポーネント	サイズ (バイト単位)
イメージヘッダー	8 バイト
イメージ記述子	24 バイト
アプリケーションバイナリ	< 1 MB - (324)
Trailer	292 バイト

イメージヘッダー

デバイス上のアプリケーションイメージは、マジックコードとイメージフラグで構成されるヘッダーで始まる必要があります。

ヘッダーフィールド	サイズ (バイト単位)
マジックコード	7 バイト
イメージフラグ	1 バイト

マジックコード

フラッシュデバイスのイメージはマジックコードで始まる必要があります。デフォルトのマジックコードは @AFRTOS です。ブートローダーは、イメージを起動する前に有効なマジックコードが存在するかどうかを確認します。これが検証の最初のステップです。

イメージフラグ

イメージフラグは、アプリケーションイメージのステータスを保存するために使用されます。フラグは OTA プロセスで使用されます。両方のバンクのイメージフラグがデバイスの状態を決定します。実行イメージがコミット保留中としてマークされている場合、デバイスが OTA 自己テストフェーズにあることを意味します。デバイス上のイメージが有効とマークされていても、起動ごとに同じ検証ステップが実行されます。イメージが新しいものとしてマークされている場合、ブートローダーはそれをコミット保留としてマークし、検証後にセルフテストのために起動します。ブートローダーはまた、ウォッチドッグタイマーを初期化して開始するため、新しい OTA イメージがセルフテストに失敗した場合、デバイスはリブートし、ブートローダーはイメージを消去して拒否し、以前の有効なイメージを実行します。

デバイスは有効なイメージを 1 つのみ持つことができます。もう 1 つのイメージは、新しい OTA イメージか、コミット保留中 (セルフテスト) です。OTA 更新が正常に完了すると、古いイメージがデバイスから消去されます。

ステータス	値	説明
新しいイメージ	0xFF	アプリケーションイメージは新しく、決して実行されません。
コミット保留中	0xFE	アプリケーションイメージにテスト実行のためのマークが付けられます。
有効です	0xFC	アプリケーションイメージが有効とマークされ、コミットされます。
無効	0xF8	アプリケーションイメージは無効とマークされています。

イメージ記述子

フラッシュデバイス上のアプリケーションイメージには、イメージヘッダーの後にイメージ記述子が含まれている必要があります。イメージ記述子は、ポストビルドユーティリティによって生成されます。ポストビルドユーティリティは、設定ファイル (`ota-descriptor.config`) を使用して適切な記述子を生成し、それをアプリケーションバイナリに付加します。このビルド後のステップの出力は、OTA に使用できるバイナリイメージです。

記述子フィールド	サイズ (バイト単位)
シーケンス番号	4 バイト
開始アドレス	4 バイト
終了アドレス	4 バイト
実行アドレス	4 バイト

記述子フィールド	サイズ (バイト単位)
ハードウェア ID	4 バイト
リザーブド	4 バイト

シーケンス番号

シーケンス番号は、新しい OTA イメージを構築する前に増加させる必要があります。`ota-descriptor.config` ファイルを参照してください。ブートローダーは、この番号を使用してブートするイメージを決定します。有効な値の範囲は 1 ~ 4294967295 です。

開始アドレス

デバイス上のアプリケーションイメージの開始アドレスです。イメージ記述子がアプリケーションバイナリの先頭に付いているため、このアドレスはイメージ記述子の先頭です。

終了アドレス

イメージトレーラーを除く、デバイス上のアプリケーションイメージの終了アドレスです。

実行アドレス

イメージの実行アドレスです。

ハードウェア ID

OTA イメージが正しいプラットフォーム用に構築されているかどうかを検証するためにブートローダーによって使用される一意のハードウェア ID です。

リザーブド

これは、将来の利用のために予約されています。

イメージトレーラー

イメージトレーラーはアプリケーションバイナリに付加されます。これには、署名タイプ文字列、署名サイズ、およびイメージの署名が含まれます。

トレーラーフィールド	サイズ (バイト単位)
署名タイプ	32 バイト

トレーラーフィールド	サイズ (バイト単位)
署名サイズ	4 バイト
署名	256 バイト

署名タイプ

署名タイプは、使用されている暗号アルゴリズムを表す文字列で、トレーラーのマーカーとして機能します。ブートローダーは、楕円曲線デジタル署名アルゴリズム (ECDSA) をサポートしています。デフォルトは sig-sha256-ecdsa です。

署名サイズ

暗号署名のサイズで、バイト単位です。

署名

イメージ記述子が付加されたアプリケーションバイナリの暗号化署名です。

ブートローダーの設定

基本的なブートローダー設定オプションは、*freertos*/vendors/microchip/boards/curiosity_pic32mzef/bootloader/config_files/aws_boot_config.h で提供されています。一部のオプションは、デバッグの目的でのみ提供されています。

デフォルトスタートを有効にする

デフォルトアドレスからアプリケーションを実行できるようにします。デバッグのためにのみ有効にする必要があります。イメージは、検証なしでデフォルトのアドレスから実行されます。

暗号署名検証を有効にする

起動時に暗号化署名検証を有効にします。失敗したイメージはデバイスから消去されます。このオプションはデバッグ目的でのみ提供されており、本番環境では有効にする必要があります。

無効なイメージを消去する

そのバンクのイメージ検証が失敗した場合に、完全なバンク消去を有効にします。このオプションはデバッグ用に提供されており、本番環境では有効にする必要があります。

ハードウェア ID 検証を有効にする

OTA イメージの記述子内のハードウェア ID と、ブートローダーにプログラムされたハードウェア ID の検証を有効にします。これはオプションで、ハードウェア ID の検証が不要な場合は無効にすることができます。

アドレス検証を有効にする

OTA イメージ記述子の開始アドレス、終了アドレス、実行アドレスの検証を有効にします。このオプションを有効にしておくことをお勧めします。

ブートローダーのビルド

デモブートローダーは、FreeRTOS ソースコードリポジトリの aws_demos にある *freertos*/vendors/microchip/boards/curiosity_pic32mzef/aws_demos/mplab/ プロジェクトに、ロード可能なプロジェクトとして含まれています。aws_demos プロジェクトがビルドされる場合、ブートローダーが最初にビルドされ、続いてアプリケーションがビルドされます。最終的な出力は、ブートローダーとアプリケーションを含む統合された 16 進数イメージです。factory_image_generator.py ユーティリティは、暗号化署名付きの統合された 16 進数イメージを生成するために提供されています。ブートローダーユーティリティのスクリプトは、*freertos*/demos/ota/bootloader/utility/ にあります。

ブートローダーのビルド前のステップ

このビルド前のステップでは、codesigner_cert_utility.py というユーティリティスクリプトが実行され、コード署名証明書からパブリックキーが抽出され、Abstract Syntax Notation One (ASN.1) エンコード形式のパブリックキーを含む C ヘッダーファイルが生成されます。このヘッダーは、ブートローダープロジェクトにコンパイルされます。生成されたヘッダーには、パブリックキーの配列とキーの長さという 2 つの定数が含まれています。ブートローダープロジェクトを aws_demos なしでビルドし、通常のアプリケーションとしてデバッグすることもできます。

AWS IoT Device Defender デモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

このデモでは、 AWS IoT Device Defender ライブラリを使用して に接続する方法を示しますAWS IoT Device Defender。このデモでは、coreMQTT ライブラリを使用して、MQTT ブローカーへの TLS 経由 AWS IoT の MQTT 接続 (相互認証) を確立し、coreJSON ライブラリを使用して、AWS IoT Device Defender サービスから受信したレスポンスを検証および解析します。このデモでは、デバイスから収集されたメトリクスを使用して JSON 形式のレポートを作成する方法と、構築されたレポートを AWS IoT Device Defender サービスに送信する方法を示します。このデモでは、coreMQTT ライブラリにコールバック関数を登録して AWS IoT Device Defender 、サービスからのレスポンスを処理し、送信されたレポートが承諾されたか拒否されたかを確認する方法も示します。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

機能

このデモでは、メトリクスを収集し、JSON 形式で Device Defender レポートを構築し、 AWS IoT MQTT ブローカーへの安全な MQTT 接続を介して AWS IoT Device Defender サービスに送信する方法を示す単一のアプリケーションタスクを作成します。デモには、標準のネットワークメトリクスとカスタムメトリクスが含まれています。カスタムメトリクスの場合、デモには以下が含まれます。

- FreeRTOS タスク ID のリストを示す「`task_numbers`」というメトリクス。このメトリクスのタイプは「数値のリスト」です。
- デモアプリケーションタスクのスタックハイウォーターマークを示す「`stack_high_water_mark`」というメトリクス。このメトリクスのタイプは「数値」です。

ネットワークメトリクスの収集方法は、使用している TCP/IP スタックによって異なります。FreeRTOS+TCP およびサポートされている lwIP 設定では、デバイスから実際のメトリクスを収集し、 AWS IoT Device Defender レポートに送信するメトリクス収集実装を提供しています。GitHub から、FreeRTOS+TCP と lwIP の実装を入手できます。

他の TCP/IP スタックを使用するボード向けに、すべてのネットワークメトリクスに対してゼロを返すメトリック収集関数のスタブ定義を提供しています。*`freertos`*`/demos/device_defender_for_aws/metrics_collector/stub/metrics_collector.c` で機能を実

装し、ネットワークスタックが実際のメトリクスを送信するようにしてください。このファイルも GitHub ウェブサイトから入手できます。

ESP32 の場合、デフォルトの lwIP 設定はコアロックを使用しないため、デモではスタブメトリクスが使用されます。参照 lwIP メトリクス収集の実装を使用する場合は、`lwiopts.h` に次のマクロを定義します。

```
#define LINK_SPEED_OF_YOUR_NETIF_IN_BPS 0
#define LWIP_TCPIP_CORE_LOCKING         1
#define LWIP_STATS                      1
#define MIB2_STATS                      1
```

以下に、デモを実行したときの出力例を示します。

```
24 1682 [iot_thread] [INFO] MQTT connection successfully established with broker.


25 1682 [iot_thread] [INFO] A clean MQTT connection is established. Cleaning up all the stored outgoing publishes.


26 1682 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/accepted.27 1682 [
iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.
28 1722 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
29 1722 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


30 1722 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/accepted with maximum QoS 1.
31 2322 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/rejected.32 2322 [
iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.
33 2382 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
34 2382 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


35 2382 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/rejected with maximum QoS 1.
36 2982 [iot_thread] [INFO] the published payload:{"hed": {"rid": 1109,"v": "1.0"},"met": {"tp": {"pts": [{"pt": 33251}],"t": 1},
"up": {"pts": [],"t": 0},"ns": {"bi": 0,"bo": 0,"pi": 0,"po": 0},"tc": {"ec": {"cs": [{"lp": 33251,"rad": "44.236.152.27:888337 2
982 [iot_thread] [INFO] PUBLISH sent for topic $aws/things/DemoThing/defender/metrics/json to broker with packet ID 3.


38 3102 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
39 3102 [iot_thread] [INFO] Ack packet deserialized with result: MQTTSuccess.
40 3102 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
41 3102 [iot_thread] [INFO] PUBACK received for packet id 3.


42 3102 [iot_thread] [INFO] Cleaned up outgoing publish packet with packet id 3.


43 3102 [iot_thread] [INFO] Packet received. ReceivedBytes=141.
44 3102 [iot_thread] [INFO] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
45 3102 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
46 3502 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.


47 3542 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
48 3542 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


49 4142 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.


50 4202 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
51 4202 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


52 4802 [iot_thread] [INFO] Disconnected from the broker.
53 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::freertos_heap::before::bytes::2088152

54 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::freertos_heap::after::bytes::1985556

55 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::demo_task_stack::before::bytes::1908

56 4802 [iot_thread] [INFO ][DEMO][4802] memory_metrics::demo_task_stack::after::bytes::1908

57 5802 [iot_thread] [INFO ][DEMO][5802] Demo completed successfully.

58 5804 [iot_thread] [INFO ][INIT][5804] SDK cleanup done.

59 5804 [iot_thread] [INFO ][DEMO][5804] -------DEMO FINISHED-------
```

ボードが FreeRTOS+TCP またはサポートされている lwIP 設定を使用していない場合、出力は次のようになります。

```
24 1716 [iot_thread] [INFO] MQTT connection successfully established with broker.


25 1716 [iot_thread] [INFO] A clean MQTT connection is established. Cleaning up all the stored outgoing publishes.


26 1716 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/accepted.27 1716
 [iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.
28 1756 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
29 1756 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


30 1756 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/accepted with maximum QoS 1.
31 2356 [iot_thread] [INFO] Attempt to subscribe to the MQTT topic $aws/things/DemoThing/defender/metrics/json/rejected.32 2356
 [iot_thread] [INFO] SUBSCRIBE sent for topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.
33 2436 [iot_thread] [INFO] Packet received. ReceivedBytes=3.
34 2436 [iot_thread] [INFO] MQTT_PACKET_TYPE_SUBACK.


35 2436 [iot_thread] [INFO] Subscribed to the topic $aws/things/DemoThing/defender/metrics/json/rejected with maximum QoS 1.
36 3036 [iot_thread] [ERROR] Using stub definition of GetNetworkStats! Please implement for your network stack to get correct m
etrics.
37 3036 [iot_thread] [ERROR] Using stub definition of GetOpenTcpPorts! Please implement for your network stack to get correct m
etrics.
38 3036 [iot_thread] [ERROR] Using stub definition of GetOpenUdpPorts! Please implement for your network stack to get correct m
etrics.
39 3036 [iot_thread] [ERROR] Using stub definition of GetEstablishedConnections! Please implement for your network stack to get
 correct metrics.
40 3036 [iot_thread] [INFO] the published payload:{"hed": {"rid": 1079,"v": "1.0"},"met": {"tp": {"pts": [],"t": 0},"up": {"pts
": [],"t": 0},"ns": {"bi": 0,"bo": 0,"pi": 0,"po": 0},"tc": {"ec": {"cs": [],"t": 0}}},"cmet": {"stack_high_water_mark": [{"num
41 3036 [iot_thread] [INFO] PUBLISH sent for topic $aws/things/DemoThing/defender/metrics/json to broker with packet ID 3.


42 3196 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
43 3196 [iot_thread] [INFO] Ack packet deserialized with result: MQTTSuccess.
44 3196 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
45 3196 [iot_thread] [INFO] PUBACK received for packet id 3.


46 3196 [iot_thread] [INFO] Cleaned up outgoing publish packet with packet id 3.


47 3196 [iot_thread] [INFO] Packet received. ReceivedBytes=141.
48 3196 [iot_thread] [INFO] De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
49 3196 [iot_thread] [INFO] State record updated. New state=MQTTPublishDone.
50 3596 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/accepted to broker.


51 3656 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
52 3656 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


53 4256 [iot_thread] [INFO] UNSUBSCRIBE sent topic $aws/things/DemoThing/defender/metrics/json/rejected to broker.


54 4336 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
55 4336 [iot_thread] [INFO] MQTT_PACKET_TYPE_UNSUBACK.


56 4936 [iot_thread] [INFO] Disconnected from the broker.
57 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::freertos_heap::before::bytes::2088152

58 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::freertos_heap::after::bytes::1985556

59 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::demo_task_stack::before::bytes::1908

60 4936 [iot_thread] [INFO ][DEMO][4936] memory_metrics::demo_task_stack::after::bytes::1908

61 5936 [iot_thread] [INFO ][DEMO][5936] Demo completed successfully.

62 5938 [iot_thread] [INFO ][INIT][5938] SDK cleanup done.

63 5938 [iot_thread] [INFO ][DEMO][5938] -------DEMO FINISHED-------
```

デモのソースコードは、*freertos*/demos/device_defender_for_aws/ ディレクトリのダウンロード内、または GitHub ウェブサイトにあります。

AWS IoT Device Defender トピックのサブスクライブ

subscribeToDefenderTopics 関数は、MQTT トピックをサブスクライブして、発行された Device Defender レポートに対する応答を受信します。ここでは、マクロ DEFENDER_API_JSON_ACCEPTED を使用して、受け入れた Device Defender レポートの応答を受信するトピック文字列を作成します。ここでは、マクロ DEFENDER_API_JSON_REJECTED を使用して、拒否した Device Defender レポートの応答を受信するトピック文字列を作成します。

デバイスメトリクスを収集する

collectDeviceMetrics 関数は、metrics_collector.h で定義されている関数を使用してネットワークメトリクスを収集します。送受信されたバイト数とパケット数、開いている TCP ポート、開いている UDP ポート、確立された TCP 接続に関するメトリクスが収集されます。

AWS IoT Device Defender レポートの生成

generateDeviceMetricsReport 関数は、report_builder.h で定義されている関数を使用して Device Defender レポートを生成します。この関数は、ネットワークメトリクスとバッファを受け取り、 が期待する形式で JSON ドキュメントを作成し AWS IoT Device Defender 、提供されたバッファに書き込みます。で想定される JSON ドキュメントの形式 AWS IoT Device Defender は、AWS IoT デベロッパーガイドの「デバイス側のメトリクス」で指定されています。

AWS IoT Device Defender レポートの発行

AWS IoT Device Defender レポートは、JSON AWS IoT Device Defender レポートを発行するための MQTT トピックで発行されます。レポートは、GitHub ウェブサイトのこのコードスニペットに示したように、マクロ DEFENDER_API_JSON_PUBLISH を使用して作成されます。

応答を処理するコールバック

publishCallback 関数は、受信する MQTT 発行メッセージを処理します。 AWS IoT Device Defender ライブラリの Defender_MatchTopic API を使用して、受信 MQTT メッセージが AWS IoT Device Defender サービスからのものであるかどうかを確認します。メッセージが AWS IoT Device Defender サービスからのものである場合、受信した JSON レスポンスを解析し、レスポンス内のレポート ID を抽出します。その後、レポート ID は、レポート内に送信されたものと同じかどうか検証されます。

AWS IoT Greengrass V1 検出デモアプリケーション

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

FreeRTOS の AWS IoT Greengrass Discovery デモを実行する前に、 AWS、 AWS IoT Greengrass、 をセットアップする必要があります AWS IoT。をセットアップするには AWS、「」の手順に従いますAWS アカウントとアクセス許可の設定。をセットアップするには AWS IoT Greengrass、Greengrass グループを作成し、Greengrass コアを追加する必要があります。設定の詳細については AWS IoT Greengrass、「 の開始方法 AWS IoT Greengrass」を参照してください。

AWS と を設定したら AWS IoT Greengrass、追加のアクセス許可を設定する必要があります AWS IoT Greengrass。

アクセス AWS IoT Greengrass 許可を設定するには

1. IAM コンソールを参照します。
2. ナビゲーションペインで [ロール] を選択した後、[Greengrass_ServiceRole] を見つけて選択します。
3. [Attach policies (ポリシーをアタッチする)]、[AmazonS3FullAccess]、[AWSIoTFullAccess]、[ポリシーのアタッチ] の順に選択します。
4. AWS IoT コンソールを参照します。
5. ナビゲーションペインで [Greengrass]、[グループ] の順に選択し、以前に作成した Greengrass グループを選びます。
6. [設定]、[ロールの追加] の順に選択します。
7. [Greengrass_ServiceRole]、[保存] の順に選択します。

ボードを に接続 AWS IoT し、FreeRTOS デモを設定します。

1. への MCU ボードの登録 AWS IoT

ボードを登録した後、新しい Greengrass ポリシーを作成し、デバイスの証明書にアタッチする必要があります。

新しい AWS IoT Greengrass ポリシーを作成するには

1. AWS IoT コンソールを参照します。
2. ナビゲーションペインで、[Secure] (保護) を選択し、[Policies] (ポリシー) を選択してから [Create] (作成) を選択します。
3. ポリシーを識別するための名前を入力します。
4. [Add statements] (ステートメントを追加) セクションで、[Advanced mode] (アドバンストモード) を選択します。次の JSON をポリシーエディタウィンドウにコピーアンドペーストします。

   ```
   {
         "Effect": "Allow",
         "Action": [
           "greengrass:*"
         ],
         "Resource": "*"
   }
   ```

   このポリシーは、すべてのリソースにアクセス AWS IoT Greengrass 許可を付与します。
5. [Create] (作成) を選択します。

デバイスの証明書に AWS IoT Greengrass ポリシーをアタッチするには

1. AWS IoT コンソールを参照します。
2. ナビゲーションペインで、[管理]、[モノ] の順に選択して、以前に作成したモノを選択します。
3. [セキュリティ] を選択し、デバイスにアタッチされている証明書を選択します。
4. [ポリシー]、[アクション]、[ポリシーのアタッチ] の順に選択します。
5. 前に作成した Greengrass ポリシーを検索して選択し、[アタッチ] を選択します。

2. FreeRTOS をダウンロードする

> **Note**
>
> FreeRTOS コンソールから FreeRTOS をダウンロードする場合は、Connect to AWS IoT Greengrass- ***Platform*** ではなく、Connect to AWS IoT- ***Platform*** を選択します。

3. FreeRTOS デモを設定する.

   freertos/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして CONFIG_GREENGRASS_DISCOVERY_DEMO_ENABLED を定義します。

と をセットアップした後 AWS IoT AWS IoT Greengrass、FreeRTOS をダウンロードして設定した後、デバイスで Greengrass デモを構築、フラッシュ、実行できます。ボードのハードウェアとソフトウェアの開発環境を設定するには、ボード固有の入門ガイド の手順に従います。

Greengrass デモは、Greengrass コアと AWS IoT MQTT クライアントに一連のメッセージを発行します。 AWS IoT MQTT クライアントでメッセージを表示するには、AWS IoT コンソールを開き、テストを選択し、MQTT テストクライアントを選択して、 にサブスクリプションを追加しますfreertos/demos/ggd。

MQTT クライアントには以下の文字列が表示されます。

```
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
```

Amazon EC2 インスタンスを使用する

Amazon EC2 インスタンスを操作している場合

1. Amazon EC2 インスタンスに関連付けられているパブリック DNS (IPv4) を見つけます。Amazon EC2 コンソールに移動し、左側のナビゲーションパネルで [Instances] (インスタンス) を選択します。Amazon EC2 インスタンスを選択し、[Description] (説明) パネルを選択します。[パブリック DNS (IPv4)] のエントリを探し、メモします。
2. [Security groups] (セキュリティグループ) のエントリを見つけて、Amazon EC2 インスタンスにアタッチされているセキュリティグループを選択します。

3. [インバウンドルール] タブを選択し、[インバウンドルールの編集] を選択して、次のルールを追加します。

   インバウンドルール

タイプ	プロトコル	ポート範囲	ソース	Description (説明) - オプション
HTTP	TCP	80	0.0.0.0/0	-
HTTP	TCP	80	::/0	-
SSH	TCP	22	0.0.0.0/0	-
カスタム TCP	TCP	8883	0.0.0.0/0	MQTT 通信
カスタム TCP	TCP	8883	::/0	MQTT 通信
HTTPS	TCP	443	0.0.0.0/0	-
HTTPS	TCP	443	::0/0	-
すべての ICMP - IPv4	ICMP	すべて	0.0.0.0/0	-
すべての ICMP - IPv4	ICMP	すべて	::0/0	-

4. AWS IoT コンソールで Greengrass、Groups を選択し、以前に作成した Greengrass グループを選択します。[設定] を選択します。[ローカル接続の検出] を [接続情報の手動管理] に変更します。
5. ナビゲーションペインで、[コア] を選択し、グループコアを選択します。
6. [接続] を選択し、コアエンドポイントが 1 つだけあり (残りのエンドポイントはすべて削除)、IP アドレスではないことを確認します (変化する可能性があるため)。最初のステップでメモしたパブリック DNS (IPv4) を使用することをお勧めします。
7. 作成した FreeRTOS IoT モノを GG グループに追加します。

   a. グループ AWS IoT Greengrass ページに戻るには、戻る矢印を選択します。ナビゲーションペインで、[デバイス] を選択し、[デバイスの追加] を選択します。

   b. [IoT モノの選択] を選択します。デバイスを選択し、[完了] を選択します。

8. 必要なサブスクリプションを追加します。 [Greengrass Group] (Greengrass グループ) ページで [Subscription] (サブスクリプション) を選択し、[Add Subscription] (サブスクリプションの追加) を選択して、ここに示す情報を入力します。

   サブスクリプション

ソース	Target	トピック
TIGG1	IoT クラウド	freertos/demos/ggd

   ここで「Source」は、ボードを登録したときに AWS IoT コンソールで作成された AWS IoT モノに付けられた名前です。ここで示す例では「TIGG1」です。

9. AWS IoT Greengrass グループのデプロイを開始し、デプロイが成功したことを確認します。これで、 AWS IoT Greengrass 検出デモを正常に実行できるようになります。

AWS IoT Greengrass V2

AWS IoT Greengrass V2 デバイスとの互換性

AWS IoT Greengrass V2 クライアントデバイスの サポートは、 と下位互換性があります AWS IoT Greengrass V1。アプリケーションコードを変更することなく FreeRTOS クライアントデバイスを V2 コアデバイスに接続できます。クライアントデバイスを V2 コアデバイスに接続できるようにするには、以下を行います。

- Greengrass ソフトウェアを Greengrass コアデバイスにデプロイします。「クライアントデバイスをコアデバイスに接続する」を参照して、デバイスを AWS IoT Greengrass V2に接続します。
- クライアントデバイス、 AWS IoT Core クラウドサービス、Greengrass コンポーネント間でメッセージ (Lambda 関数を含む) を中継するには、MQTT ブリッジコンポーネントをデプロイして設定します。
- IP ディテクタコンポーネントをデプロイして接続情報を自動検出するか、エンドポイントを手動で管理します。
- 詳細については、「ローカル AWS IoT デバイスを操作する」を参照してください。

詳細については、 AWS IoT Greengrass V1 でのアプリケーション AWS IoT Greengrass V2の実行に関する AWS ドキュメントを参照してください。

coreHTTP のデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

これらのデモは、coreHTTP ライブラリの使い方を学習するのに役立ちます。

トピック

- coreHTTP 相互認証のデモ
- coreHTTP 基本 Amazon S3 アップロードのデモ
- coreHTTP 基本 S3 ダウンロードのデモ
- coreHTTP 基本マルチスレッドのデモ

coreHTTP 相互認証のデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

coreHTTP (相互認証) デモプロジェクトでは、クライアントとサーバー間で TLS と相互認証を使用して HTTP サーバーへの接続を確立する方法を示します。このデモでは、mbedTLS ベースのトランスポートインターフェイス実装を使用して、サーバーおよびクライアントに認証された TLS 接続を確立します。また、HTTP でのリクエストレスポンスワークフローを示します。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

機能

このデモでは、以下のステップを実行する方法を示す単一のアプリケーションタスクと例を作成します。

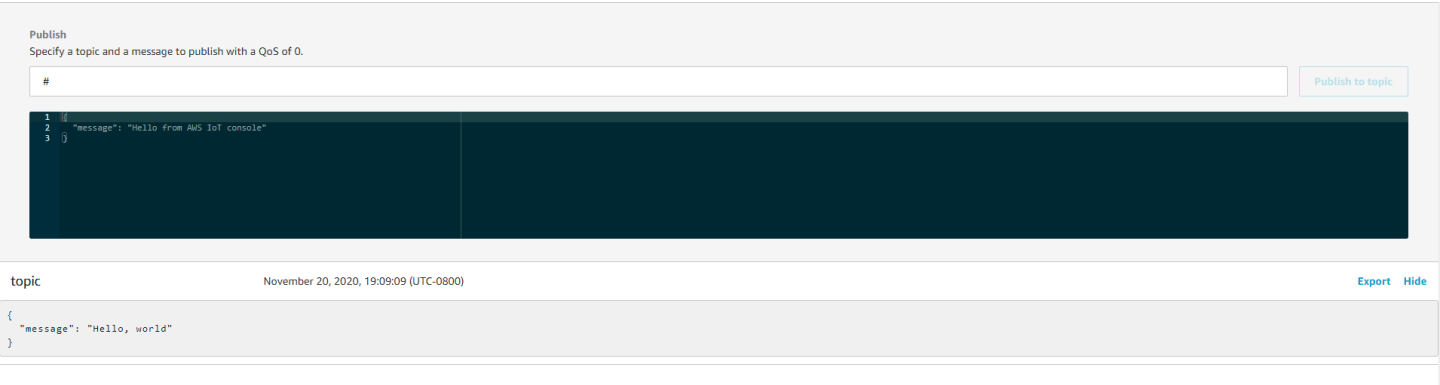
- AWS IoT エンドポイントの HTTP サーバーに接続します。
- POST リクエストを送信する。
- レスポンスを受信する。
- サーバーから切断する。

これらのステップを完了すると、このデモによって以下のスクリーンショットのような出力が生成されます。

```
9 1565 [iot_thread] [INFO ][DEMO][1565] ---------STARTING DEMO---------

10 1566 [iot_thread] [INFO ][INIT][1566] SDK successfully initialized.
11 1566 [iot_thread] [INFO ][DEMO][1566] Successfully initialized the demo. Network type for the demo: 4
12 1566 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:319] 13 1566 [iot_thread] Establishing a TLS session to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com:8443.14 1566 [iot_thread]
15 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (35.166.102.88) will be stored
16 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (35.166.2.12) will be stored
17 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (44.239.154.164) will be stored
18 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (44.239.97.33) will be stored
19 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (44.238.43.70) will be stored
20 1622 [iot_thread] DNS[0x68F5]: The answer to 'a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com' (52.32.144.134) will be stored
21 2042 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:393] 22 2042 [iot_thread] Sending HTTP POST request to a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com/topics/topic?qos=1...23 2042 [iot_thread]
24 2082 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:418] 25 2082 [iot_thread] Received HTTP response from a2zk5tjv9x07ct-ats.iot.us-west-2.amazonaws.com/topics/topic?qos=1...
26 2082 [iot_thread]
27 2082 [iot_thread] [INFO] [HTTPDemo] [http_demo_mutual_auth.c:283] 28 2082 [iot_thread] Demo completed successfully.29 2082 [iot_thread]
30 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::freertos_heap::before::bytes::2088152
31 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::freertos_heap::after::bytes::1990104
32 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::demo_task_stack::before::bytes::1908
33 2082 [iot_thread] [INFO ][DEMO][2082] memory_metrics::demo_task_stack::after::bytes::1908
34 3082 [iot_thread] [INFO ][DEMO][3082] Demo completed successfully.
35 3084 [iot_thread] [INFO ][INIT][3084] SDK cleanup done.
36 3084 [iot_thread] [INFO ][DEMO][3084] -------DEMO FINISHED-------
```

AWS IoT コンソールは、次のスクリーンショットのような出力を生成します。



ソースコードの編成

デモソースファイルの名前は `http_demo_mutual_auth.c` です。このファイルは、*freertos*/`demos/coreHTTP/` ディレクトリと GitHub ウェブサイトにあります。

HTTP AWS IoT サーバーへの接続

connectToServerWithBackoffRetries 関数は、HTTP サーバーへの相互認証された TLS AWS IoT 接続を試みます。接続が失敗すると、タイムアウト後に接続を再試行します。タイムアウト値は、最大試行回数に達するか、最大タイムアウト値に達するまで、指数関数的に増加します。`RetryUtils_BackoffAndSleep` 関数は、指数関数的に増加するタイムアウト値を提供し、最大試行回数に達したときに `RetryUtilsRetriesExhausted` を返します。`connectToServerWithBackoffRetries` 関数は、設定された試行回数に達してもブローカーへの TLS 接続を確立できない場合に、失敗ステータスを返します。

HTTP リクエストの送信とレスポンスの受信

prvSendHttpRequest 関数は、HTTP サーバーに POST AWS IoT リクエストを送信する方法を示しています。で REST API にリクエストを行う方法の詳細については AWS IoT、「デバイス通信プロトコル - HTTPS」を参照してください。レスポンスは、同じ coreHTTP API 呼び出しである `HTTPClient_Send` で受信されます。

coreHTTP 基本 Amazon S3 アップロードのデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

この例では、Amazon Simple Storage Service (Amazon S3) HTTP サーバーに PUT リクエストを送信し、小さなファイルをアップロードする方法を示します。また、アップロード後に GET リクエストを実行して、ファイルのサイズを確認します。この例で使用するネットワークトランスポートインターフェイスは、mbedTLS を使用して、coreHTTP を実行する IoT デバイスクライアントと Amazon S3 HTTP サーバー間に相互認証された接続を確立します。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

シングルスレッドとマルチスレッド

coreHTTP には、シングルスレッドとマルチスレッド (マルチタスク) の 2 つの使用モデルがあります。このデモは、このセクションでは 1 つのスレッドで HTTP ライブラリを実行しますが、実際にはシングルスレッド環境で coreHTTP を使用する方法を示しています。このデモでは、1 つのタスクのみが HTTP API を使用します。シングルスレッドアプリケーションは HTTP ライブラリを繰り返し呼び出す必要がありますが、マルチスレッドアプリケーションは、代わりにエージェント (またはデーモン) タスク内でバックグラウンドで HTTP リクエストを送信できます。

ソースコードの編成

デモソースファイルの名前は `http_demo_s3_upload.c` です。このファイルは、*freertos*`/demos/coreHTTP/` ディレクトリと GitHub ウェブサイトにあります。

Amazon S3 HTTP サーバー接続の設定

このデモでは、署名付き URL を使用して Amazon S3 HTTP サーバーに接続し、ダウンロードするオブジェクトへのアクセスを許可します。Amazon S3 HTTP サーバーの TLS 接続では、サーバー認証のみが使用されます。アプリケーションレベルでは、オブジェクトへのアクセスは、署名付き URL クエリのパラメータを使用して認証されます。以下のステップに従って、 AWSへの接続を設定します。

1. AWS アカウントをセットアップします。
    a. まだ作成していない場合は、 AWS アカウントを作成します。
    b. アカウントとアクセス許可は AWS Identity and Access Management (IAM) を使用して設定されます。IAM は、アカウント内の各ユーザーのアクセス権限の管理に使用します。デフォルトでは、ルート所有者によって付与されるまで、ユーザーにはアクセス権限がありません。
        i. AWS アカウントにユーザーを追加するには、「IAM ユーザーガイド」を参照してください。
        ii. このポリシー AWS IoT を追加して、FreeRTOS および にアクセスするためのアクセス許可を AWS アカウントに付与します。

- AmazonS3FullAccess

2. Amazon Simple Storage Service ユーザーガイドの S3 バケットを作成する方法に従って、Amazon S3 にバケットを作成します。
3. S3 バケットにファイルとフォルダをアップロードする方法のステップに従って、Amazon S3 にファイルをアップロードします。
4. `FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/presigned_urls_gen.py` ファイルにあるスクリプトを使用して、署名付き URL を生成します。

   使用手順については、`FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/README.md` ファイルを参照してください。

機能

このデモでは、まず TLS サーバー認証を使用して Amazon S3 HTTP サーバーに接続します。次に、HTTP リクエストを作成して、`democonfigDEMO_HTTP_UPLOAD_DATA` に指定されているデータをアップロードします。ファイルをアップロードした後、ファイルのサイズをリクエストして、ファイルが正常にアップロードされたことを確認します。デモのソースコードは GitHub ウェブサイトに記載されています。

Amazon S3 HTTP サーバーに接続する

connectToServerWithBackoffRetries 関数は HTTP サーバーへの TCP 接続を試みます。接続が失敗すると、タイムアウト後に接続を再試行します。タイムアウト値は、最大試行回数に達するか、最大タイムアウト値に達するまで、指数関数的に増加します。`connectToServerWithBackoffRetries` 関数は、設定された試行回数に達してもサーバーへの TCP 接続を確立できない場合に、失敗ステータスを返します。

`prvConnectToServer` 関数は、サーバー認証のみを使用して Amazon S3 HTTP サーバーへの接続を確立する方法を示します。この関数は、`FreeRTOS-Plus/Source/Application-Protocols/network_transport/freertos_plus_tcp/using_mbedtls/using_mbedtls.c` ファイルで実装されている mbedTLS ベースのインターフェイスを使用します。`prvConnectToServer` の定義は、GitHub ウェブサイトに記載されています。

データをアップロードする

`prvUploadS3ObjectFile` 関数は、PUT リクエストの作成方法とアップロードするファイルの指定方法を示します。ファイルのアップロード先の Amazon S3 バケットと、アップロードするファイ

ルの名前は、署名付き URL に指定されます。メモリを節約するために、リクエストヘッダーとレスポンスの受信の両方に同じバッファが使用されます。レスポンスは、HTTPClient_Send API 関数を使用して同期的に受信されます。200 OK レスポンスステータスコードが Amazon S3 HTTP サーバーから返されることが想定されます。その他のステータスコードはエラーです。

prvUploadS3ObjectFile() のソースコードは、GitHub ウェブサイトに記載されています。

アップロードを検証する

prvVerifyS3ObjectFileSize 関数は、prvGetS3ObjectFileSize を呼び出して S3 バケットのオブジェクトのサイズを取得します。Amazon S3 HTTP サーバーは、現在署名付き URL を使用した HEAD リクエストをサポートしていないため、0 番目のバイトがリクエストされます。ファイルのサイズは、レスポンスの Content-Range ヘッダーフィールドに含まれています。206 Partial Content レスポンスがサーバーから返されることが想定されます。その他のレスポンスステータスコードはエラーです。

prvGetS3ObjectFileSize() のソースコードは、GitHub ウェブサイトに記載されています。

coreHTTP 基本 S3 ダウンロードのデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

このデモでは、範囲リクエストを使用して Amazon S3 HTTP サーバーからファイルをダウンロードする方法を示します。範囲リクエストは、HTTPClient_AddRangeHeader を使用して HTTP リクエストを作成するときに coreHTTP API でネイティブにサポートされます。マイクロコントローラ環境では、範囲リクエストを使用することを強くお勧めします。単一のリクエストはなく、個別の複数の範囲リクエストで大きなファイルをダウンロードすることで、ネットワークソケットをブロックすることなくファイルの各セクションを処理できます。範囲リクエストを使用すると、パケットがドロップされて TCP 接続での再送が必要になるリスクが低くなるため、デバイスの消費電力が改善されます。

この例で使用するネットワークトランスポートインターフェイスは、mbedTLS を使用して、coreHTTP を実行する IoT デバイスクライアントと Amazon S3 HTTP サーバー間に相互認証された接続を確立します。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

シングルスレッドとマルチスレッド

coreHTTP には、シングルスレッドとマルチスレッド (マルチタスク) の 2 つの使用モデルがあります。このデモは、このセクションでは 1 つのスレッドで HTTP ライブラリを実行しますが、実際にはシングルスレッド環境で coreHTTP を使用する方法を示しています (このデモでは、1 つのタスクのみが HTTP API を使用します)。シングルスレッドアプリケーションは HTTP ライブラリを繰り返し呼び出す必要がありますが、マルチスレッドアプリケーションは、代わりにエージェント (またはデーモン) タスク内でバックグラウンドで HTTP リクエストを送信できます。

ソースコードの編成

デモプロジェクトの名前は `http_demo_s3_download.c` です。このプロジェクトは、*`freertos`*`/demos/coreHTTP/` ディレクトリと GitHub ウェブサイトにあります。

Amazon S3 HTTP サーバー接続の設定

このデモでは、署名付き URL を使用して Amazon S3 HTTP サーバーに接続し、ダウンロードするオブジェクトへのアクセスを許可します。Amazon S3 HTTP サーバーの TLS 接続では、サーバー認証のみが使用されます。アプリケーションレベルでは、オブジェクトへのアクセスは、署名付き URL クエリのパラメータを使用して認証されます。以下のステップに従って、 AWSへの接続を設定します。

1. AWS アカウントをセットアップします。
   a. まだ作成していない場合は、 AWS アカウントを作成してアクティブ化します。
   b. アカウントとアクセス許可は AWS Identity and Access Management (IAM) を使用して設定されます。IAM では、アカウント内の各ユーザーのアクセス権限を管理できます。デフォルトでは、ルート所有者によって付与されるまで、ユーザーにはアクセス権限がありません。

i. AWS アカウントにユーザーを追加するには、「IAM ユーザーガイド」を参照してください。

ii. これらのポリシー AWS IoT を追加して、FreeRTOS および にアクセスするためのアクセス許可を AWS アカウントに付与します。

- AmazonS3FullAccess

2. Amazon Simple Storage Service Console ユーザーガイドの S3 バケットを作成する方法に従って、S3 にバケットを作成します。
3. S3 バケットにファイルとフォルダをアップロードする方法のステップに従って、S3 にファイルをアップロードします。
4. `FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/presigned_urls_gen.py` ファイルにあるスクリプトを使用して、署名付き URL を生成します。使用手順については、「`FreeRTOS-Plus/Demo/coreHTTP_Windows_Simulator/Common/presigned_url_generator/README.md`」を参照してください。

機能

このデモは、まずファイルのサイズを取得します。次に、各バイト範囲をループ内で順番にリクエストします。範囲サイズは `democonfigRANGE_REQUEST_LENGTH` で指定します。

デモのソースコードは GitHub ウェブサイトに記載されています。

Amazon S3 HTTP サーバーに接続する

connectToServerWithBackoffRetries() 関数は HTTP サーバへの TCP 接続を試みます。接続が失敗すると、タイムアウト後に接続を再試行します。タイムアウト値は、最大試行回数に達するか、最大タイムアウト値に達するまで、指数関数的に増加します。`connectToServerWithBackoffRetries()` は、設定された試行回数に達してもサーバーへの TCP 接続を確立できない場合に、失敗ステータスを返します。

`prvConnectToServer()` 関数は、サーバー認証のみを使用して Amazon S3 HTTP サーバーへの接続を確立する方法を示します。この関数は、FreeRTOS-Plus/Source/Application-Protocols/network_transport/freertos_plus_tcp/using_mbedtls/using_mbedtls.c ファイルに実装されている mbedTLS ベースのトランスポートインターフェイスを使用します。

`prvConnectToServer()` のソースコードは、GitHub に記載されています。

範囲リクエストを作成する

HTTPClient_AddRangeHeader() API 関数は、バイト範囲を HTTP リクエストヘッダーにシリアル化して、範囲リクエストを形成することをサポートします。このデモでは、範囲リクエストを使用して、ファイルサイズを取得し、ファイルの各セクションをリクエストします。

prvGetS3ObjectFileSize() 関数は、S3 バケットのファイルのサイズを取得します。Amazon S3 へのこの最初のリクエストには、レスポンスの送信後も接続を開いたままにするために Connection: keep-alive ヘッダーが追加されます。S3 HTTP サーバーは、現在署名付き URL を使用した HEAD リクエストをサポートしていないため、0 番目のバイトがリクエストされます。ファイルのサイズは、レスポンスの Content-Range ヘッダーフィールドに含まれています。206 Partial Content レスポンスがサーバーから返されることが想定されます。受信するその他のレスポンスステータスコードはエラーです。

prvGetS3ObjectFileSize() のソースコードは、GitHub に記載されています。

このデモは、ファイルサイズを取得した後、ダウンロードするファイルのバイト範囲ごとに新しい範囲リクエストを作成します。ファイルの各セクションについて HTTPClient_AddRangeHeader() を使用します。

範囲リクエストの送信とレスポンスの受信

prvDownloadS3ObjectFile() 関数は、ファイル全体がダウンロードされるまで、範囲リクエストをループ内で送信します。HTTPClient_Send() API 関数は、リクエストの送信とレスポンスの受信を同期的に行います。関数からのレスポンスは xResponse で受信されます。その後、ステータスコードが 206 Partial Content であることが検証され、これまでにダウンロードされたバイト数の増加が Content-Length ヘッダー値で表されます。

prvDownloadS3ObjectFile() のソースコードは、GitHub に記載されています。

coreHTTP 基本マルチスレッドのデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

このデモでは、FreeRTOS のスレッドセーフキューを使用して、処理を待機しているリクエストとレスポンスを保持します。このデモでは、3 つのタスクに注目してください。

- メインタスクは、リクエストがリクエストキューに表示されるのを待ちます。表示されたリクエストをネットワーク経由で送信し、レスポンスをレスポンスキューに配置します。
- リクエストタスクは、サーバーに送信する HTTP ライブラリリクエストオブジェクトを作成し、リクエストキューに配置します。各リクエストオブジェクトには、アプリケーションにダウンロードするように設定されている S3 ファイルのバイト範囲を指定します。
- レスポンスタスクは、レスポンスがレスポンスキューに表示されるのを待ちます。受信したすべてのレスポンスをログに記録します。

この基本マルチスレッドのデモは、サーバー認証のみを使用する TLS 接続を使用するように設定されています。これは Amazon S3 HTTP サーバーの要件です。アプリケーションレイヤー認証は、署名付き URL クエリの署名バージョン 4 パラメータを使用して実行されます。

ソースコードの編成

デモプロジェクトの名前は `http_demo_s3_download_multithreaded.c` です。このプロジェクトは、*`freertos`*`/demos/coreHTTP/` ディレクトリと GitHub ウェブサイトにあります。

デモプロジェクトを構築する

このデモプロジェクトでは、Visual Studio の無料のコミュニティエディションを使用します。次の手順でデモを構築します。

1. Visual Studio IDE 内から `mqtt_multitask_demo.sln` Visual Studio ソリューションファイルを開きます。
2. IDE の [Build] (構築) メニューから [Build Solution] (ソリューションの構築) を選択します。

> **Note**
>
> Microsoft Visual Studio 2017 以前を使用している場合は、お使いのバージョンと互換性があるプラットフォームツールセットを選択する必要があります ([Project] (プロジェクト) -> [RTOSDemos Properties] (RTOSDemos プロパティ) -> [Platform Toolset] (プラットフォームツールセット))。

デモプロジェクトを設定する

このデモでは、FreeRTOS+TCP TCP/IP スタックを使用します。TCP/IP スタータープロジェクトの指示に従って、次の手順を実行します。

1. 前提条件コンポーネント (WinPCap など) をインストールします。
2. オプションで静的または動的 IP アドレス、ゲートウェイアドレス、ネットマスクを設定します。
3. オプションで MAC アドレスを設定します。
4. ホストマシンのイーサネットネットワークインターフェイスを選択します。
5. 重要: HTTP デモを実行する前にネットワーク接続をテストします。

Amazon S3 HTTP サーバー接続の設定

coreHTTP 基本ダウンロードのデモの Amazon S3 HTTP サーバー接続の設定 の指示に従います。

機能

このデモでは、合計 3 つのタスクを作成します。

- リクエストを送信し、ネットワーク経由でレスポンスを受信するタスク。
- 送信するリクエストを作成するタスク。
- 受信したレスポンスを処理するタスク。

このデモのメインタスクの動作:

1. リクエストキューとレスポンスキューを作成します。
2. サーバーへの接続を作成します。
3. リクエストタスクとレスポンスタスクを作成します。
4. リクエストキューがネットワーク経由でリクエストを送信するのを待ちます。
5. ネットワーク経由で受信したレスポンスをレスポンスキューに配置します。

リクエストタスクの動作:

1. 各範囲リクエストを作成します。

レスポンスタスクの動作:

1. 受信した各レスポンスを処理します。

Typedefs

このデモでは、マルチスレッドをサポートするために次の構造を定義しています。

リクエスト項目

次の構造で、リクエストキューに配置するリクエスト項目を定義します。リクエスト項目は、リクエストタスクが HTTP リクエストを作成した後、キューにコピーされます。

```
/**
 * @brief Data type for the request queue.
 *
 * Contains the request header struct and its corresponding buffer, to be
 * populated and enqueued by the request task, and read by the main task. The
 * buffer is included to avoid pointer inaccuracy during queue copy operations.
 */
typedef struct RequestItem
{
    HTTPRequestHeaders_t xRequestHeaders;
    uint8_t ucHeaderBuffer[ democonfigUSER_BUFFER_LENGTH ];
} RequestItem_t;
```

レスポンス項目

次の構造で、レスポンスキューに配置するレスポンス項目を定義します。レスポンス項目は、メインの HTTP タスクがネットワーク経由でレスポンスを受信した後、キューにコピーされます。

```
/**
 * @brief Data type for the response queue.
 *
 * Contains the response data type and its corresponding buffer, to be enqueued
 * by the main task, and interpreted by the response task. The buffer is
 * included to avoid pointer inaccuracy during queue copy operations.
 */
typedef struct ResponseItem
```

```
{
    HTTPResponse_t xResponse;
    uint8_t ucResponseBuffer[ democonfigUSER_BUFFER_LENGTH ];
} ResponseItem_t;
```

メインの HTTP 送信タスク

メインのアプリケーションタスクの動作:

1. ホストアドレスの署名付き URL を解析して、Amazon S3 HTTP サーバーとの接続を確立します。
2. S3 バケット内のオブジェクトへのパスの署名付き URL を解析します。
3. サーバー認証と TLS を使用して Amazon S3 HTTP サーバーに接続します。
4. リクエストキューとレスポンスキューを作成します。
5. リクエストタスクとレスポンスタスクを作成します。

`prvHTTPDemoTask()` 関数でこのセットアップを行い、デモのステータスを返します。この関数のソースコードは、GitHub に記載されています。

`prvDownloadLoop()` 関数では、メインタスクがリクエストキューからのリクエストをブロックして待機します。リクエストを受信すると、API 関数 `HTTPClient_Send()` を使用してリクエストを送信します。この API 関数が成功すると、レスポンスをレスポンスキューに配置します。

`prvDownloadLoop()` のソースコードは、GitHub に記載されています。

HTTP リクエストタスク

リクエストタスクは `prvRequestTask` 関数で指定されます。この関数のソースコードは、GitHub に記載されています。

リクエストタスクは、Amazon S3 バケット内のファイルのサイズを取得します。これは `prvGetS3ObjectFileSize` 関数で実行されます。Amazon S3 へのこのリクエストには、レスポンスの送信後も接続を開いたままにするために「Connection: keep-alive」ヘッダーが追加されます。Amazon S3 HTTP サーバーは、現在署名付き URL を使用した HEAD リクエストをサポートしていないため、0 番目のバイトがリクエストされます。ファイルのサイズは、レスポンスの `Content-Range` ヘッダーフィールドに含まれています。`206 Partial Content` レスポンスがサーバーから返されることが想定されます。受信するその他のレスポンスステータスコードはエラーです。

prvGetS3ObjectFileSize のソースコードは、GitHub に記載されています。

ファイルサイズを取得した後も、リクエストタスクはファイルの各範囲のリクエストを続けます。各範囲リクエストはリクエストキューに配置され、メインタスクによって送信されます。ファイル範囲は、デモユーザーがマクロ democonfigRANGE_REQUEST_LENGTH で設定します。範囲リクエストは、HTTPClient_AddRangeHeader 関数を使用するときに HTTP クライアントライブラリ API でネイティブにサポートされます。prvRequestS3ObjectRange 関数は、HTTPClient_AddRangeHeader() の使用方法を示します。

prvRequestS3ObjectRange 関数のソースコードは、GitHub に記載されています。

HTTP レスポンスタスク

レスポンスタスクは、ネットワーク経由で受信されたレスポンスのレスポンスキューで待ちます。メインタスクは、HTTP レスポンスを正常に受信するとレスポンスキューに配置します。このタスクは、ステータスコード、ヘッダー、および本文をログに記録してレスポンスを処理します。実世界のアプリケーションは、例えばレスポンス本体をフラッシュメモリに書き込んでレスポンスを処理できます。レスポンスステータスコードが 206 partial content でない場合、このタスクはデモが失敗することをメインタスクに通知します。レスポンスタスクは prvResponseTask 関数で指定されます。この関数のソースコードは、GitHub に記載されています。

AWS IoT ジョブライブラリのデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

AWS IoT Jobs ライブラリデモでは、MQTT 接続を介して AWS IoT Jobs サービスに接続し、 からジョブを取得し AWS IoT、デバイスで処理する方法を示します。 AWS IoT Jobs デモプロジェクトは FreeRTOS Windows ポートを使用するため、Windows の Visual Studio コミュニティバージョンで構築して評価できます。マイクロコントローラハードウェアは必要ありません。このデモでは、と同じ方法で TLS を使用して AWS IoT MQTT ブローカーへの安全な接続を確立しますcoreMQTT Mutual Authentication デモ。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

ソースコードの編成

デモのコードは `jobs_demo.c` ファイルにあり、GitHub ウェブサイトまたは *`freertos`*`/demos/jobs_for_aws/` ディレクトリで見つけることができます。

AWS IoT MQTT ブローカー接続を設定する

このデモでは、MQTT ブローカーへの AWS IoT MQTT 接続を使用します。この接続は、coreMQTT Mutual Authentication デモ と同じ方法で設定されています。

機能

このデモでは、 からジョブを受信 AWS IoT してデバイスで処理するために使用されるワークフローを示します。デモはインタラクティブであり、 AWS IoT コンソールまたは AWS Command Line Interface () を使用してジョブを作成する必要がありますAWS CLI。ジョブの作成の詳細については、「AWS CLI コマンドリファレンス」の「create-job」を参照してください。デモでは、メッセージをコンソールに出力するために、`action` キーが `print` に設定されたジョブドキュメントが必要です。

このジョブドキュメントについては、次の形式を参照してください。

```
{
    "action": "print",
    "message": "ADD_MESSAGE_HERE"
}
```

次のコマンド例のように AWS CLI 、 を使用してジョブを作成できます。

```
aws iot create-job \
    --job-id t12 \
    --targets arn:aws:iot:region:123456789012:thing/device1 \
    --document '{"action":"print","message":"hello world!"}'
```

デモでは、メッセージをトピックに再発行するために、action キーが publish に設定されたジョブドキュメントも必要です。このジョブドキュメントについては、次の形式を参照してください。

```
{
    "action": "publish",
    "message": "ADD_MESSAGE_HERE",
    "topic": "topic/name/here"
}
```

デモは、デモを終了するために action キーが exit に設定されたジョブドキュメントを受信するまでループします。ジョブドキュメントの形式は次のとおりです。

```
{
    "action: "exit"
}
```

ジョブデモのエントリポイント

ジョブデモのエントリポイント関数のソースコードは、GitHub にあります。この関数は次の操作を実行します。

1. mqtt_demo_helpers.c のヘルパー関数を使用して MQTT 接続を確立します。
2. NextJobExecutionChanged のヘルパー関数を使用して mqtt_demo_helpers.c API 向けの MQTT トピックをサブスクライブします。トピック文字列は、 AWS IoT ジョブライブラリで定義されたマクロを使用して、前にアセンブルされます。
3. StartNextPendingJobExecution のヘルパー関数を使用して mqtt_demo_helpers.c API 向けの MQTT トピックを発行します。トピック文字列は、 AWS IoT ジョブライブラリで定義されたマクロを使用して、前にアセンブルされます。
4. MQTT_ProcessLoop を繰り返し呼び出して着信メッセージを受信し、処理のために prvEventCallback に渡します。
5. デモが終了アクションを受け取ったら、mqtt_demo_helpers.c ファイルのヘルパー関数を使用して、MQTT トピックのサブスクライブを解除し、接続を切断します。

受信した MQTT メッセージのコールバック

prvEventCallback 関数は AWS IoT Jobs ライブラリJobs_MatchTopicから を呼び出して、受信 MQTT メッセージを分類します。メッセージタイプが新しいジョブに対応している場合、prvNextJobHandler() が呼び出されます。

prvNextJobHandler 関数とこの関数に呼び出される関数は、JSON 形式のメッセージからジョブドキュメントを解析し、ジョブで指定されたアクションを実行します。特に興味深いのは `prvSendUpdateForJob` 関数です。

実行中のジョブの更新を送信する

prvSendUpdateForJob() 関数は、ジョブライブラリから `Jobs_Update()` を呼び出し、直後の MQTT 公開操作で使用されるトピック文字列を入力します。

coreMQTT デモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

以下のデモでは、coreMQTT ライブラリの使い方について説明します。

トピック

- coreMQTT Mutual Authentication デモ
- coreMQTT エージェント接続共有デモ

coreMQTT Mutual Authentication デモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

coreMQTT Mutual Authentication デモプロジェクトでは、クライアントとサーバー間の相互認証で TLS を使用して MQTT ブローカーへの接続を確立する方法を説明します。このデモでは mbedTLS

ベースのトランスポートインターフェイスの実装を使用して、サーバーとクライアント間の認証 TLS 接続を確立し、QoS 1 レベルでの MQTT のサブスクライブおよび公開ワークフローについて説明します。トピックフィルターをサブスクライブし、フィルターに一致するトピックに公開し、QoS 1 レベルでサーバからそれらのメッセージを受信するまで待機します。ブローカーに公開し、ブローカーから同じメッセージを受信するこのサイクルは、無限に繰り返されます。このデモのメッセージは QoS 1 で送信され、MQTT 仕様に従って少なくとも 1 つの配信が保証されます。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

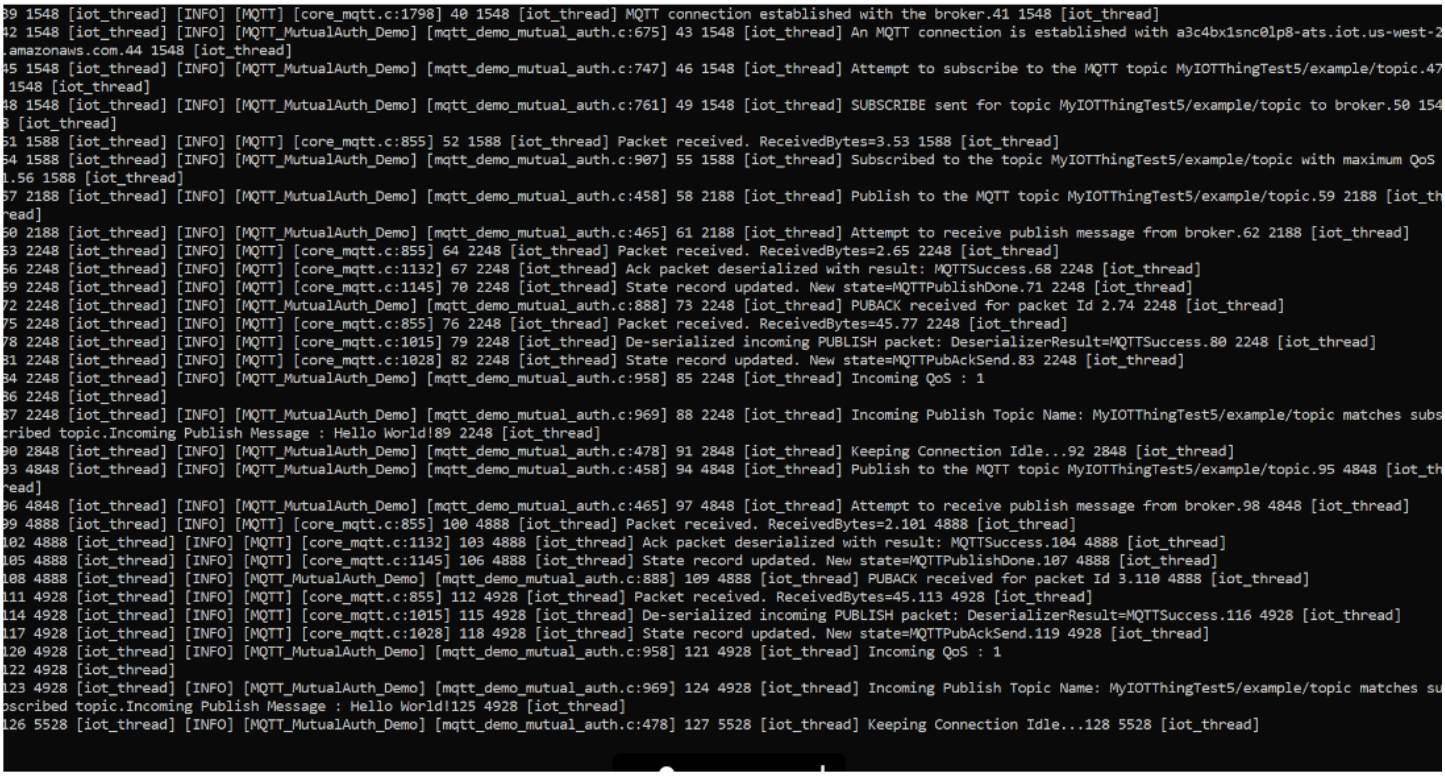
ソースコード

デモのソースファイルの名前は `mqtt_demo_mutual_auth.c` で、*`freertos`*`/demos/coreMQTT/` ディレクトリと GitHub ウェブサイトで手に入ります。

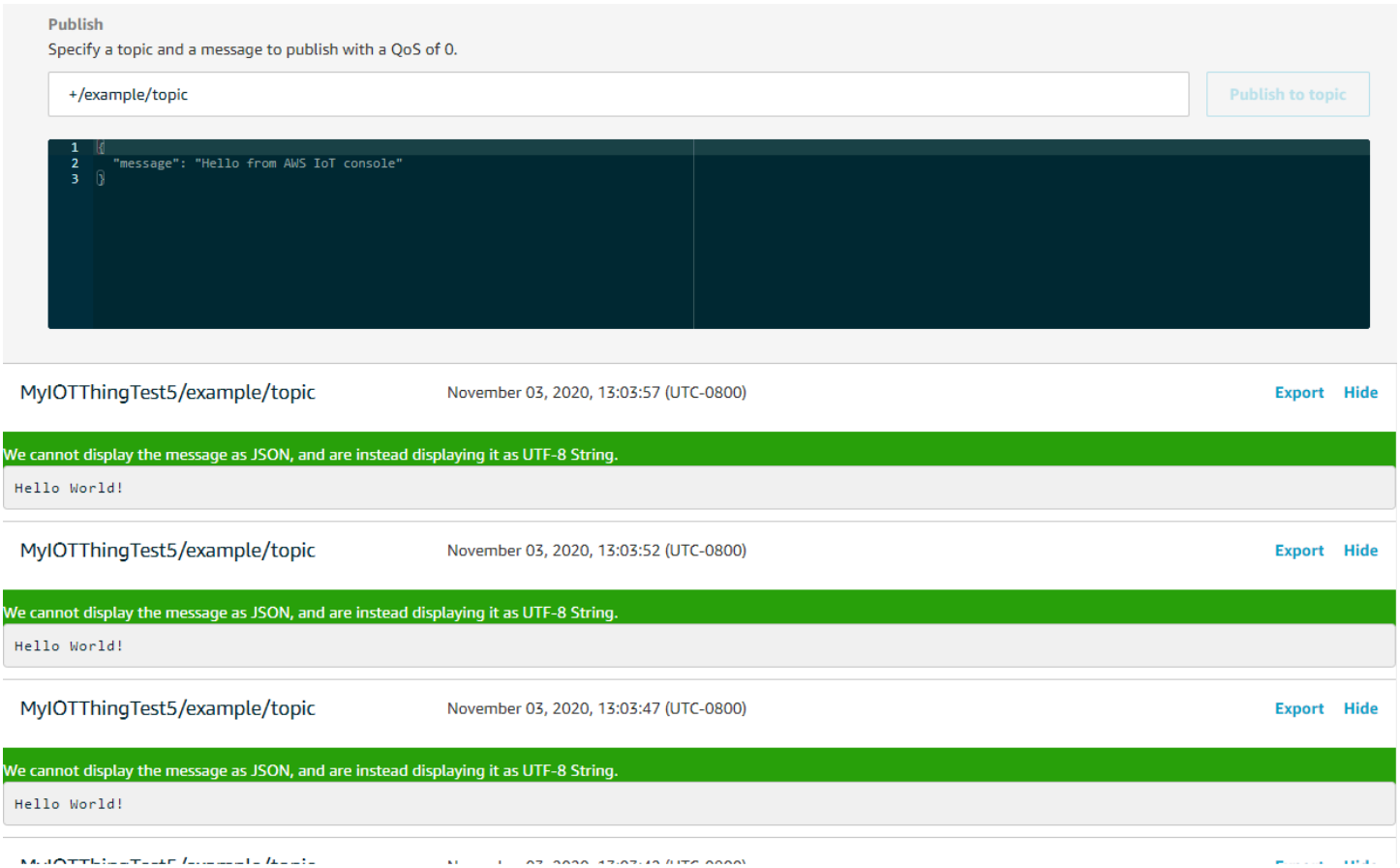
機能

デモでは、ブローカーに接続し、ブローカーのトピックをサブスクライブし、ブローカーのトピックに公開し、最後にブローカーから切断する方法を示す一連の例をループ処理する 1 つのアプリケーションタスクを作成します。デモアプリケーションは、同じトピックをサブスクライブし、公開します。デモが MQTT ブローカーにメッセージを公開するたびに、ブローカーは同じメッセージをデモアプリケーションに送り返します。

デモが正常に完了すると、次の図のような出力が生成されます。



AWS IoT コンソールは、次のイメージのような出力を生成します。



エクスポネンシャルバックオフとジッターを使用してロジックを再試行する

prvBackoffForRetry 関数は、TLS 接続や MQTT サブスクライブ要求など、サーバーでの失敗したネットワークオペレーションを、エクスポネンシャルバックオフとジッターでどのように再試行できるかを示します。この関数は次の再試行のバックオフ時間を計算し、再試行回数がまだ残っている場合にバックオフ遅延を実行します。バックオフ時間の計算には乱数の生成が必要なため、この関数は PKCS11 モジュールを使用して乱数を生成します。PKCS11 モジュールを使用すると、ベンダーのプラットフォームで真性乱数生成器 (TRNG) がサポートされている場合、真性乱数生成器にアクセスできます。接続再試行中のデバイスからの衝突の可能性を軽減するため、乱数生成器にデバイス固有のエントロピーソースをシードすることをお勧めします。

MQTT ブローカーへの接続

prvConnectToServerWithBackoffRetries 関数は、MQTT ブローカーへの相互認証された TLS 接続の確立を試みます。接続が失敗すると、バックオフ時間の後に再試行されます。バックオフ時間は最大試行回数に達するか、最大バックオフ時間に達するまで、指数関数的に増加します。`BackoffAlgorithm_GetNextBackoff` 関数は指数関数的に増加するバックオフ値を提供し、最大試行回数に達した場合に `RetryUtilsRetriesExhausted` を返しま

す。prvConnectToServerWithBackoffRetries 関数は、設定された試行回数に達してもブローカーへの TLS 接続を確立できない場合に、失敗ステータスを返します。

prvCreateMQTTConnectionWithBroker 関数は、クリーンセッションで MQTT ブローカーへの MQTT 接続を確立する方法を示します。これは TLS トランスポートインターフェイスを使用し、TLS トランスポートインターフェイスは FreeRTOS-Plus/Source/Application-Protocols/platform/freertos/transport/src/tls_freertos.c ファイルで実装されます。ブローカーのキープアライブ (秒) は xConnectInfo で設定することに注意してください。

次の関数は、TLS トランスポートインターフェイスと time 関数が MQTT_Init 関数を使って MQTT コンテキストでどのように設定されるかを示します。また、イベントコールバック関数ポインタ (prvEventCallback) の設定方法も表しています。このコールバックは、受信メッセージのレポートに使用されます。

MQTT トピックのサブスクライブ

prvMQTTSubscribeWithBackoffRetries 関数は、MQTT ブローカーのトピックフィルターをサブスクライブする方法を示しています。この例では、1 つのトピックフィルターをサブスクライブする方法について説明しますが、同じサブスクライブ API コールでトピックフィルターのリストを渡して、複数のトピックフィルターをサブスクラブすることも可能です。また、MQTT ブローカーがサブスクリプション要求を拒否した場合、RETRY_MAX_ATTEMPTS に関してサブスクリプションはエクスポネンシャルバックオフで再試行します。

トピックへの公開

prvMQTTPublishToTopic 関数は、MQTT ブローカーのトピックに公開する方法を示しています。

受信メッセージの受信

アプリケーションは前述のように、ブローカーに接続する前にイベントコールバック関数を登録します。prvMQTTDemoTask 関数は MQTT_ProcessLoop 関数を呼び出して受信メッセージを受信します。受信 MQTT メッセージを受信すると、受信 MQTT メッセージはアプリケーションが登録したイベントコールバック関数を呼び出します。prvEventCallback 関数は、このようなイベントコールバック関数の一例です。prvEventCallback は受信パケットタイプを調べ、適切なハンドラを呼び出します。以下の例では、関数は受信した発行メッセージを処理する prvMQTTProcessIncomingPublish()、確認 (ACK) を処理する prvMQTTProcessResponse() のいずれかを呼び出します。

受信 MQTT 公開パケットの処理

prvMQTTProcessIncomingPublish 関数は、MQTT ブローカーから公開パケットを処理する方法を示しています。

トピックからのサブスクリプションの解除

ワークフローの最後のステップは、ブローカーが mqttexampleTOPIC から発行されたメッセージを送信しないように、トピックのサブスクライブを解除することです。prvMQTTUnsubscribeFromTopic 関数の定義は次のとおりです。

デモで使用するルート CA の変更

デフォルトでは、FreeRTOS デモは Amazon ルート CA 1 証明書 (RSA 2048 ビットキー) を使用して AWS IoT Core サーバーで認証します。Amazon Root CA 3 証明書 (ECC 256 ビットキー) など、他のサーバー認証用の CA 証明書も使用できます。coreMQTT Mutual Authentication デモのルート CA を変更するには、次の手順を実行します。

1. テキストエディタで、*freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/mqtt_demo_mutual_auth_config.h ファイルを開きます。
2. ファイルで、次の行を見つけます。

   ```
   * #define democonfigROOT_CA_PEM    "...insert here..."
   ```

   この行のコメントを解除し、必要に応じてコメントブロックの末尾 */ まで移動させます。
3. 使用する CA 証明書をコピーし、"...insert here..." テキストに貼り付けます。結果は次の例のようになります。

   ```
   #define democonfigROOT_CA_PEM   "-----BEGIN CERTIFICATE-----\n"\
   "MIIBtjCCAVugAwIBAgITBmyf1XSXNmY/Owua2eiedgPySjAKBggqhkjOPQQDAjA5\n"\
   "MQswCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6b24g\n"\
   "Um9vdCBDQSAzMB4XDTE1MDUyNjAwMDAwMFoXDTQwMDUyNjAwMDAwMFowOTELMAkG\n"\
   "A1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJvb3Qg\n"\
   "Q0EgMzBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABCmXp8ZBf8ANm+gBG1bG8lKl\n"\
   "ui2yEujSLtf6ycXYqm0fc4E7O5hrOXwzpcVOho6AF2hiRVd9RFgdszflZwjrZt6j\n"\
   "QjBAMA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/BAQDAgGGMB0GA1UdDgQWBBSr\n"\
   "ttvXBp43rDCGB5Fwx5zEGbF4wDAKBggqhkjOPQQDAgNJADBGAiEA4IWSoxe3jfkr\n"\
   "BqWTrBqYaGFy+uGh0PsceGCmQ5nFuMQCIQCcAu/xlJyzlvnrxir4tiz+OpAUFteM\n"\
   "YyRIHN8wfdVoOw==\n"\
   "-----END CERTIFICATE-----\n"
   ```

4. (オプション) 他のデモのルート CA を変更できます。*freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/*demo-name*_config.h ファイルごとに、ステップ 1〜3 を繰り返します。

coreMQTT エージェント接続共有デモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

coreMQTT 接続共有デモプロジェクトでは、マルチスレッドアプリケーションを使用して、クライアントとサーバー間の相互認証で TLS を使用して AWS MQTT ブローカーへの接続を確立する方法を示します。このデモでは、mbedTLS ベースのトランスポートインターフェイスの実装を使用して、サーバーとクライアントの認証 TLS 接続を確立し、QoS 1 レベルでの MQTT のサブスクライブおよび公開ワークフローについて説明します。デモでは、トピックフィルターをサブスクライブし、フィルターに一致するトピックに公開し、QoS 1 レベルでサーバからそれらのメッセージを受信するまで待機します。ブローカーに発行し、ブローカーから同じメッセージを受信するというこのサイクルは、作成されたタスクごとに何度も繰り返されます。このデモのメッセージは QoS 1 で送信され、MQTT 仕様に従って少なくとも 1 つの配信が保証されます。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

このデモでは、スレッドセーフなキューを使用して MQTT API を操作するためのコマンドを保持します。このデモでは、注意すべきタスクが 2 つあります。

- MQTT エージェント (メイン) タスクは、コマンドキューからのコマンドを処理し、他のタスクがそれらをキューに追加します。このタスクはループに入り、その間にコマンドキューからのコマンドを処理します。終了コマンドを受信すると、このタスクはループから抜け出します。

- demo subpub タスクは MQTT トピックへのサブスクリプションを作成後に、発行オペレーションを作成してコマンドキューにプッシュします。その後、これらの発行オペレーションが MQTT エージェントタスクによって実行されます。demo subpub タスクは、コマンド完了コールバックの実行に伴い通知される発行の完了まで待機してから、次の発行を開始する前に短い遅延に入ります。このタスクでは、アプリケーションタスクが coreMQTT エージェント API を使用する方法の例を示します。

発行メッセージを着信した場合、coreMQTT エージェントは単一のコールバック関数を呼び出します。このデモには、サブスクライブしているトピックで着信した発行メッセージを呼び出すコールバックをタスクで指定できるサブスクリプションマネージャも含まれています。このデモでエージェントが着信した発行コールバックは、サブスクリプションマネージャを呼び出し、サブスクリプションを登録したタスクに発行をファンアウトします。

このデモでは、相互認証を使用した TLS 接続を使用して AWSに接続します。デモ中にネットワークが予期せず切断された場合、クライアントはエクスポネンシャルバックオフロジックを使用して再接続を試みます。クライアントが正常に再接続してもブローカーが前回のセッションを再開できない場合、クライアントは前回のセッションと同じトピックに再度サブスクライブします。

シングルスレッドとマルチスレッド

coreMQTT の使用モデルは、シングルスレッドとマルチスレッド (マルチタスク) の 2 つがあります。シングルスレッドモデルでは、coreMQTT ライブラリを 1 つのスレッドからだけ使用するため、MQTT ライブラリで明示的に繰り返し呼び出す必要があります。マルチスレッドユースケースでは、ここで説明しているデモに示すように、エージェント (またはデーモン) タスク内で MQTT プロトコルをバックグラウンドで実行できます。エージェントタスクで MQTT プロトコルを実行する場合、MQTT 状態を明示的に管理したり、`MQTT_ProcessLoop` API 関数を呼び出したりする必要はありません。また、エージェントタスクを使用すると、ミューテックスなどの同期プリミティブを必要とせずに、複数のアプリケーションタスクで単一の MQTT 接続を共有できます。

ソースコード

デモソースファイルは、`mqtt_agent_task.c` と `simple_sub_pub_demo.c` という名前で、GitHub ウェブサイトの *`freertos`*`/demos/coreMQTT_Agent/` ディレクトリに用意されています。

機能

このデモでは、少なくとも 2 つのタスクを作成します。それは MQTT API コールのリクエストを処理するプライマリタスクと、それらのリクエストを作成する設定可能な数のサブタスクです。この

デモでは、プライマリタスクがサブタスクを作成し、処理ループを呼び出し、後でクリーンアップします。プライマリタスクは、サブタスク間で共有されるブローカーへの MQTT 接続を 1 つ作成します。サブタスクはブローカーに対する MQTT サブスクリプションを作成し、それに宛ててメッセージを発行します。各サブタスクでは、発行に固有のトピックが使用されます。

メインタスク

主なアプリケーションタスクである RuncoreMqtTagentDemo は、MQTT セッションを確立し、サブタスクを作成し、終了コマンドを受信するまで処理ループ mqtTagent_commandLoop を実行します。ネットワークが予期せず切断された場合、デモはバックグラウンドでブローカーに再接続し、ブローカーに対するサブスクリプションを再確立します。処理ループが終了すると、デモはブローカーから切断されます。

コマンド

coreMQTT エージェント API を呼び出すと、エージェントタスクのキューに送信されるコマンドが作成され、MQTTAgent_CommandLoop() で処理されます。コマンドの作成時に、オプションの完了コールバックおよびコンテキストパラメータを渡すこともできます。対応するコマンドが完了すると、渡されたコンテキストと、コマンドの結果として作成された戻り値を使用して、完了コールバックが呼び出されます。完了コールバックの署名は次のとおりです。

```
typedef void (* MQTTAgentCommandCallback_t )( void * pCmdCallbackContext,
                                               MQTTAgentReturnInfo_t * pReturnInfo );
```

コマンド完了コンテキストはユーザー定義であり、このデモでは、struct MQTTAgentCommandContext となります。

コマンドは次の時点で完了したと見なされます。

- QoS > 0 でのサブスクライブ、サブスクライブ解除、発行: 対応する確認応答パケットを受信した時点。
- その他すべてのオペレーション: 対応する coreMQTT API が呼び出された時点。

発行情報、サブスクリプション情報、完了コンテキストなど、コマンドで使用されるすべての構成要素は、コマンドが完了するまでスコープ内に留めておく必要があります。呼び出し元のタスクは、完了コールバックを呼び出す前に、コマンドの構成要素を再利用してはいけません。完了コールバックは MQTT エージェントによって呼び出されるため、コマンドを作成したタスクではなく、エージェントタスクのスレッドコンテキストを使用して実行されます。タスク通知やキューなどのプロセス間通信メカニズムを使用すると、呼び出し元のタスクにコマンドの完了を通知できます。

コマンドループの実行

コマンドは MQTTAgent_CommandLoop() で継続的に処理されます。処理するコマンドがない場合、ループはコマンドがキューに追加されるまで最大で MQTT_AGENT_MAX_EVENT_QUEUE_WAIT_TIME 待機します。コマンドが追加されない場合は、MQTT_ProcessLoop() の反復が 1 回実行されます。これにより、MQTT キープアライブが管理され、キューにコマンドがない場合でも着信する発行を受け入れられます。

コマンドループ関数は、次の理由で値を返します。

- コマンドは MQTTSuccess 以外のすべてのステータスコードを返します。エラーステータスはコマンドループによって返されるため、エラーステータスについては処理方法を決定できます。このデモでは、TCP 接続が再確立され、再接続が試行されます。エラーが発生した場合、MQTT を使用している他のタスクの介入なしに、バックグラウンドで再接続することができます。
- 切断コマンド (MQTTAgent_Disconnect から) が処理されます。TCP を切断できるように、コマンドループが終了します。
- 終了コマンド (MQTTAgent_Terminate から) が処理されます。このコマンドは、キュー内に残っているコマンドや、確認応答パケットを待っているコマンドをエラーとしてマークし、コード MQTTRecvFailed を返します。

サブスクリプションマネージャ

デモでは複数のトピックを使用するため、サブスクリプションマネージャは、サブスクライブされたトピックを一意のコールバックまたはタスクに関連付けるのに役立ちます。このデモのサブスクリプションマネージャはシングルスレッドであるため、複数のタスクで同時に使用しないでください。このデモのサブスクリプションマネージャ関数は MQTT エージェントに渡されるコールバック関数からのみ呼び出され、エージェントタスクのスレッドコンテキストでのみ実行されます。

シンプルなサブスクライブ発行タスク

prvSimpleSubscribePublishTask の各インスタンスは、MQTT トピックへのサブスクリプションを作成し、そのトピックに対する発行オペレーションを作成します。複数の発行タイプを示すために、偶数番のタスクでは QoS 0 (発行パケットの送信時に完了) が使用され、奇数番のタスクでは QoS 1 (PUBACK パケットの受信時に完了) が使用されます。

無線通信経由更新デモアプリケーション

FreeRTOS には、無線通信経由 (over-the-air、OTA) ライブラリの機能を示すデモアプリケーションが含まれています。OTA デモアプリケーションは、*freertos*/demos/ota/

ota_demo_core_mqtt/ota_demo_core_mqtt.c または *freertos*/demos/ota/ota_demo_core_http/ota_demo_core_http.c のファイルにあります。

OTA デモアプリケーションは、次の操作を実行します。

1. FreeRTOS ネットワークスタックおよび MQTT バッファプールを初期化します。
2. vRunOTAUpdateDemo() を使用して OTA ライブラリを実行するタスクを作成します。
3. _establishMqttConnection() を使用して MQTT クライアントを作成します。
4. を使用して AWS IoT MQTT ブローカーに接続IotMqtt_Connect()し、MQTT 切断コールバック を登録しますprvNetworkDisconnectCallback。
5. OTA タスクを作成するために OTA_AgentInit() を呼び出し、OTA タスクが完了したときに使用されるコールバックを登録します。
6. xOTAConnectionCtx.pvControlClient = _mqttConnection; との MQTT 接続を再利用します。
7. MQTT が切断されると、アプリケーションは OTA エージェントを一時停止し、ジッターを伴う指数関数的な遅延を使用して再接続を試行し、OTA エージェントを再開します。

OTA 更新を使用する前に、FreeRTOS 無線通信経由更新 のすべての前提条件を満たしてください。

OTA 更新の設定を完了したら、OTA 機能をサポートするプラットフォームで FreeRTOS OTA デモをダウンロード、構築、フラッシュ、実行します。次の FreeRTOS 対応デバイスでは、デバイス固有のデモの手順が利用できます。

- Texas Instruments CC3220SF-LAUNCHXL
- Microchip Curiosity PIC32MZEF
- Espressif ESP32
- Renesas RX65N での FreeRTOS OTA デモのダウンロード、構築、フラッシュ、および実行

デバイスで OTA デモアプリケーションを構築、フラッシュ、実行したら、 AWS IoT コンソールまたは AWS CLI を使用して OTA 更新ジョブを作成できます。OTA 更新ジョブを作成した後、ターミナルエミュレーターを接続して OTA 更新の進行状況を表示します。プロセス中に生成されたエラーを書きとめておきます。

OTA 更新ジョブが正常に行われると、次のような出力が表示されます。この例では、簡潔にするために一部の行がリストから削除されています。

```
    249 21207 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    250 21247 [MQTT Agent Task] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
 ReceivedBytes=601.
    251 21247 [MQTT Agent Task] [core_mqtt.c:1045] [INFO] [MQTT] De-serialized incoming
 PUBLISH packet: DeserializerResult=MQTTSuccess.
    252 21248 [MQTT Agent Task] [core_mqtt.c:1058] [INFO] [MQTT] State record updated.
 New state=MQTTPubAckSend.
    253 21249 [MQTT Agent Task] [ota_demo_core_mqtt.c:976] [INFO] [MQTT] Received job
 message callback, size 548.
    254 21252 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobId: AFR_OTA-9702f1a3-b747-4c3e-a0eb-a3b0cf83ddbb]
    255 21253 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.streamname: AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f]
    256 21255 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.protocols: ["MQTT"]]
    257 21256 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[filepath: aws_demos.bin]
    258 21257 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[filesize: 1164016]
    259 21258 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileid: 0]
    260 21259 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[certfile: ecdsa-sha256-signer.crt.pem]
    261 21260 [OTA Agent Task] [ota.c:1575] [INFO] [OTA] Extracted parameter [ sig-
sha256-ecdsa: MEQCIE1SFkIHHiZAvkPpu6McJtx7SYoD... ]
    262 21261 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileType: 0]
    263 21262 [OTA Agent Task] [ota.c:2199] [INFO] [OTA] Job document was accepted.
 Attempting to begin the update.
    264 21263 [OTA Agent Task] [ota.c:2323] [INFO] [OTA] Job parsing success:
 OtaJobParseErr_t=OtaJobParseErrNone, Job name=AFR_OTA-9702f1a3-b747-4c3e-a0eb-
a3b0cf83ddbb
    265 21318 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    266 21418 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    267 21469 [OTA Agent Task] [ota.c:938] [INFO] [OTA] Setting OTA data interface.
    268 21470 [OTA Agent Task] [ota.c:2839] [INFO] [OTA] Current State=[CreatingFile],
 Event=[ReceivedJobDocument], New state=[CreatingFile]
    269 21482 [MQTT Agent Task] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
 ReceivedBytes=3.
```

```
    270 21483 [OTA Agent Task] [ota_demo_core_mqtt.c:1503] [INFO] [MQTT] SUBSCRIBED
 to topic $aws/things/__test_infra_thing71/streams/AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f/data/cbor to bro
    271 21484 [OTA Agent Task] [ota.c:2839] [INFO] [OTA] Current
 State=[RequestingFileBlock], Event=[CreateFile], New state=[RequestingFileBlock]
    272 21518 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
 Queued: 0   Processed: 0   Dropped: 0
    273 21532 [MQTT Agent Task] [core_mqtt_agent_command_functions.c:76] [INFO] [MQTT]
 Publishing message to $aws/things/__test_infra_thing71/streams/AFR_OTA-945d320b-
a18b-441b-b435-4a18d4e7671f/
    274 21534 [OTA Agent Task] [ota_demo_core_mqtt.c:1553] [INFO] [MQTT] Sent PUBLISH
 packet to broker $aws/things/__test_infra_thing71/streams/AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f/get/cbor
    275 21534 [OTA Agent Task] [ota_mqtt.c:1112] [INFO] [OTA] Published to MQTT
 topic to request the next block: topic=$aws/things/__test_infra_thing71/streams/
AFR_OTA-945d320b-a18b-441b-b435-4a1
    276 21537 [OTA Agent Task] [ota.c:2839] [INFO] [OTA] Current
 State=[WaitingForFileBlock], Event=[RequestFileBlock], New state=[WaitingForFileBlock]
    277 21558 [MQTT Agent Task] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
 ReceivedBytes=4217.
    278 21559 [MQTT Agent Task] [core_mqtt.c:1045] [INFO] [MQTT] De-serialized incoming
 PUBLISH packet: DeserializerResult=MQTTSuccess.
    279 21560 [MQTT Agent Task] [core_mqtt.c:1058] [INFO] [MQTT] State record updated.
 New state=MQTTPublishDone.
    280 21561 [MQTT Agent Task] [ota_demo_core_mqtt.c:1026] [INFO] [MQTT] Received data
 message callback, size 4120.
    281 21563 [OTA Agent Task] [ota.c:2464] [INFO] [OTA] Received valid file block:
 Block index=0, Size=4096
    282 21566 [OTA Agent Task] [ota.c:2683] [INFO] [OTA] Number of blocks remaining:
 284

    ... // Output removed for brevity

    3672 42745 [OTA Agent Task] [ota.c:2464] [INFO] [OTA] Received valid file block:
 Block index=284, Size=752
    3673 42747 [OTA Agent Task] [ota.c:2633] [INFO] [OTA] Received final block of the
 update.
    (428298) ota_pal: No such certificate file: ecdsa-sha256-signer.crt.pem. Using
 certificate in ota_demo_config.h.
    3674 42818 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 285
 Queued: 285   Processed: 284   Dropped: 0
    3675 42918 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 285
 Queued: 285   Processed: 284   Dropped: 0
```

```
   ... // Output removed for brevity

   3678 43197 [OTA Agent Task] [ota.c:2654] [INFO] [OTA] Received entire update and
validated the signature.
   3685 43215 [OTA Agent Task] [ota_demo_core_mqtt.c:862] [INFO] [MQTT] Received
OtaJobEventActivate callback from OTA Agent.

   ... // Output removed for brevity

   2 39 [iot_thread] [INFO ][DEMO][390] ---------STARTING DEMO---------

   [0;32mI (3633) WIFI: WIFI_EVENT_STA_CONNECTED
   [0;32mI (4373) WIFI: SYSTEM_EVENT_STA_GOT_IP

   ... // Output removed for brevity

   4 351 [sys_evt] [INFO ][DEMO][3510] Connected to WiFi access point, ip address:
255.255.255.0.
   5 351 [iot_thread] [INFO ][DEMO][3510] Successfully initialized the demo. Network
type for the demo: 1
   6 351 [iot_thread] [ota_demo_core_mqtt.c:1902] [INFO] [MQTT] OTA over MQTT demo,
Application version 0.9.1
   7 351 [iot_thread] [ota_demo_core_mqtt.c:1323] [INFO] [MQTT] Creating a TLS
connection to <endpoint>-ats.iot.us-west-2.amazonaws.com:8883.
   9 718 [iot_thread] [core_mqtt.c:886] [INFO] [MQTT] Packet received.
ReceivedBytes=2.
   10 718 [iot_thread] [core_mqtt_serializer.c:970] [INFO] [MQTT] CONNACK session
present bit not set.
   11 718 [iot_thread] [core_mqtt_serializer.c:912] [INFO] [MQTT] Connection accepted.

   ... // Output removed for brevity

   17 736 [OTA Agent Task] [ota_demo_core_mqtt.c:1503] [INFO] [MQTT] SUBSCRIBED to
topic $aws/things/__test_infra_thing71/jobs/notify-next to broker.
   18 737 [OTA Agent Task] [ota_mqtt.c:381] [INFO] [OTA] Subscribed to MQTT topic:
$aws/things/__test_infra_thing71/jobs/notify-next
   30 818 [iot_thread] [ota_demo_core_mqtt.c:1850] [INFO] [MQTT]  Received: 0
Queued: 0   Processed: 0   Dropped: 0
   31 819 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
value]=[execution.jobId: AFR_OTA-9702f1a3-b747-4c3e-a0eb-a3b0cf83ddbb]
   32 820 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
value]=[execution.statusDetails.updatedBy: 589824]
```

```
    33 822 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.streamname: AFR_OTA-945d320b-a18b-441b-
b435-4a18d4e7671f]
    34 823 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[execution.jobDocument.afr_ota.protocols: ["MQTT"]]
    35 824 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[filepath: aws_demos.bin]
    36 825 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[filesize: 1164016]
    37 826 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileid: 0]
    38 827 [OTA Agent Task] [ota.c:1645] [INFO] [OTA] Extracted parameter: [key:
 value]=[certfile: ecdsa-sha256-signer.crt.pem]
    39 828 [OTA Agent Task] [ota.c:1575] [INFO] [OTA] Extracted parameter [ sig-sha256-
ecdsa: MEQCIE1SFkIHHiZAvkPpu6McJtx7SYoD... ]
    40 829 [OTA Agent Task] [ota.c:1684] [INFO] [OTA] Extracted parameter: [key:
 value]=[fileType: 0]
    41 830 [OTA Agent Task] [ota.c:2102] [INFO] [OTA] In self test mode.
    42 830 [OTA Agent Task] [ota.c:1936] [INFO] [OTA] New image has a higher version
 number than the current image: New image version=0.9.1, Previous image version=0.9.0
    43 832 [OTA Agent Task] [ota.c:2120] [INFO] [OTA] Image version is valid: Begin
 testing file: File ID=0
    53 896 [OTA Agent Task] [ota.c:794] [INFO] [OTA] Beginning self-test.
    62 971 [OTA Agent Task] [ota_demo_core_mqtt.c:1553] [INFO] [MQTT] Sent PUBLISH
 packet to broker $aws/things/__test_infra_thing71/jobs/AFR_OTA-9702f1a3-b747-4c3e-
a0eb-a3b0cf83ddbb/update to br63 971 [MQTT Agent Task] [core_mqtt.c:1045] [INFO] [MQTT]
 De-serialized incoming PUBLISH packet: DeserializerResult=MQTTSuccess.
    65 973 [MQTT Agent Task] [core_mqtt.c:1058] [INFO] [MQTT] State record updated. New
 state=MQTTPublishDone.
    64 973 [OTA Agent Task] [ota_demo_core_mqtt.c:902] [INFO] [MQTT] Successfully
 updated with the new image.
```

無線通信経由デモ設定

OTA デモ設定は、aws_iot_ota_update_demo.c で提供されるデモ固有の設定オプションです。これらの設定は、OTA ライブラリ設定ファイルで提供されている OTA ライブラリ設定とは異なります。

OTA_DEMO_KEEP_ALIVE_SECONDS

MQTT クライアントの場合、この設定は、1 つの制御パケットの送信が完了してから次の制御パケットの送信を開始するまでの最大時間間隔です。制御パケットがない場合、PINGREQ が送信されます。ブローカーは、このキープアライブ間隔の 1 倍半の時間でメッセージまたは

PINGREQ パケットを送信しないクライアントを切断する必要があります。この設定は、アプリケーションの要件に基づいて調整する必要があります。

OTA_DEMO_CONN_RETRY_BASE_INTERVAL_SECONDS

ネットワーク接続を再試行するまでの基本間隔 (秒単位)。OTA デモは、この基本時間間隔の後に再接続を試みます。この間隔は、試行が失敗するたびに倍増します。この基本遅延の最大値までのランダムな遅延も間隔に追加されます。

OTA_DEMO_CONN_RETRY_MAX_INTERVAL_SECONDS

ネットワーク接続を再試行するまでの最大間隔 (秒単位)。再接続遅延は試行が失敗するたびに倍増しますが、この最大値に同じ間隔のジッターを加えた値を超えることはありません。

Texas Instruments CC3220SF-LAUNCHXL での FreeRTOS OTA デモのダウンロード、構築、フラッシュ、実行

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

FreeRTOS と OTA デモコードをダウンロードするには

- GitHub サイト (https://github.com/FreeRTOS/FreeRTOS) からソースコードをダウンロードすることができます。

デモアプリケーションを構築するには

1. の手順に従ってFreeRTOS の使用を開始する、aws_demosプロジェクトを Code Composer Studio にインポートし、 AWS IoT エンドポイント、Wi-Fi SSID とパスワード、ボードのプライベートキーと証明書を設定します。
2. *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメント

アウトして CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。

3. ソリューションを構築し、エラーなしでビルドされていることを確認します。
4. ターミナルエミュレーターを起動し、次の設定を使用してボードに接続します。
   - ボーレート: 115200
   - データビット: 8
   - パリティ: なし
   - ストップビット: 1
5. ボード上でプロジェクトを実行し、Wi-Fi と AWS IoT MQTT メッセージブローカーに接続できることを確認します。

実行時に、ターミナルエミュレーターは次のようなテキストを表示するはずです。

```
    0 1000 [Tmr Svc] Simple Link task created
    Device came up in Station mode
    1 2534 [Tmr Svc] Write certificate...
    2 5486 [Tmr Svc] [ERROR] Failed to destroy object. PKCS11_PAL_DestroyObject failed.
    3 5486 [Tmr Svc] Write certificate...
    4 5776 [Tmr Svc] Security alert threshold = 15
    5 5776 [Tmr Svc] Current number of alerts = 1
    6 5778 [Tmr Svc] Running Demos.
    7 5779 [iot_thread] [INFO ][DEMO][5779] ---------STARTING DEMO---------
    8 5779 [iot_thread] [INFO ][INIT][5779] SDK successfully initialized.
    Device came up in Station mode
    [WLAN EVENT] STA Connected to the AP: afrlab-pepper , BSSID: 74:83:c2:b4:46:27
    [NETAPP EVENT] IP acquired by the device
    Device has connected to afrlab-pepper
    Device IP Address is 192.168.36.176
    9 8283 [iot_thread] [INFO ][DEMO][8282] Successfully initialized the demo. Network
 type for the demo: 1
    10 8283 [iot_thread] [INFO] OTA over MQTT demo, Application version 0.9.0
    11 8283 [iot_thread] [INFO] Creating a TLS connection to <endpoint>-ats.iot.us-
west-2.amazonaws.com:8883.
    12 8852 [iot_thread] [INFO] Creating an MQTT connection to <endpoint>-ats.iot.us-
west-2.amazonaws.com.
    13 8914 [iot_thread] [INFO] Packet received. ReceivedBytes=2.
    14 8914 [iot_thread] [INFO] CONNACK session present bit not set.
    15 8914 [iot_thread] [INFO] Connection accepted.
    16 8914 [iot_thread] [INFO] Received MQTT CONNACK successfully from broker.
```

```
    17 8914 [iot_thread] [INFO] MQTT connection established with the broker.
    18 8915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
    19 8953 [OTA Agent T] [INFO] Current State=[RequestingJob], Event=[Start], New
 state=[RequestingJob]
    20 9008 [MQTT Agent ] [INFO] Packet received. ReceivedBytes=3.
    21 9015 [OTA Agent T] [INFO] SUBSCRIBED to topic $aws/things/__test_infra_thing73/
jobs/notify-next to broker.
    22 9015 [OTA Agent T] [INFO] Subscribed to MQTT topic: $aws/things/
__test_infra_thing73/jobs/notify-next
    23 9504 [MQTT Agent ] [INFO] Publishing message to $aws/things/
__test_infra_thing73/jobs/$next/get.
    24 9535 [MQTT Agent ] [INFO] Packet received. ReceivedBytes=2.
    25 9535 [MQTT Agent ] [INFO] Ack packet deserialized with result: MQTTSuccess.
    26 9536 [MQTT Agent ] [INFO] State record updated. New state=MQTTPublishDone.
    27 9537 [OTA Agent T] [INFO] Sent PUBLISH packet to broker $aws/things/
__test_infra_thing73/jobs/$next/get to broker.
    28 9537 [OTA Agent T] [WARN] OTA Timer handle NULL for Timerid=0, can't stop.
    29 9537 [OTA Agent T] [INFO] Current State=[WaitingForJob],
 Event=[RequestJobDocument], New state=[WaitingForJob]
    30 9539 [MQTT Agent ] [INFO] Packet received. ReceivedBytes=120.
    31 9539 [MQTT Agent ] [INFO] De-serialized incoming PUBLISH packet:
 DeserializerResult=MQTTSuccess.
    32 9540 [MQTT Agent ] [INFO] State record updated. New state=MQTTPublishDone.
    33 9540 [MQTT Agent ] [INFO] Received job message callback, size 62.
    34 9616 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution
    35 9616 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobId
    36 9617 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobDocument
    37 9617 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobDocument.afr_ota
    38 9617 [OTA Agent T] [INFO] Failed job document content
 check: Required job document parameter was not extracted:
 parameter=execution.jobDocument.afr_ota.protocols
    39 9618 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=execution.jobDocument.afr_ota.files
    40 9618 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=filesize
    41 9618 [OTA Agent T] [INFO] Failed job document content check: Required job
 document parameter was not extracted: parameter=fileid
    42 9619 [OTA Agent T] [INFO] Failed to parse JSON document as AFR_OTA job:
 DocParseErr_t=7
```

```
   43 9619 [OTA Agent T] [INFO] No active job available in received job document:
OtaJobParseErr_t=OtaJobParseErrNoActiveJobs
   44 9619 [OTA Agent T] [ERROR] Failed to execute state transition handler: Handler
returned error: OtaErr_t=OtaErrJobParserError
   45 9620 [OTA Agent T] [INFO] Current State=[WaitingForJob],
Event=[ReceivedJobDocument], New state=[CreatingFile]
   46 9915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   47 10915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   48 11915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   49 12915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   50 13915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
   51 14915 [iot_thread] [INFO]  Received: 0   Queued: 0   Processed: 0   Dropped: 0
```

Microchip Curiosity PIC32MZEF での FreeRTOS OTA デモのダウンロード、構築、フラッシュ、実行

Important

このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

Note

Microchip との合意に基づき、Curiosity PIC32MZEF (DM320104) は FreeRTOS Reference Integration リポジトリのメインブランチから削除されるため、新しいリリースには搭載されません。Microchip は PIC32MZEF (DM320104) が新しい設計に推奨されなくなったことを公式に発表しました。PIC32MZEF プロジェクトとソースコードには、以前のリリースタグから引き続きアクセスできます。Microchip では、新しい設計に Curiosity PIC32MZ-EF-2.0 Development board (DM320209) を使用するよう推奨しています。Pic32mzv1 プラットフォームは、FreeRTOS リファレンス統合リポジトリの v202012.00 に引き続き用意されています。ただし、プラットフォームは FreeRTOS リファレンスの v202107.00 によってサポートされなくなりました。

FreeRTOS OTA デモコードをダウンロードするには

- GitHub サイト (https://github.com/FreeRTOS/FreeRTOS) からソースコードをダウンロードすることができます。

OTA 更新デモアプリケーションを構築するには

1. の手順に従ってFreeRTOS の使用を開始する、MPLAB X IDE にaws_demosプロジェクトをインポートし、 AWS IoT エンドポイント、Wi-Fi SSID とパスワード、ボードのプライベートキーと証明書を設定します。
2. vendors/*vendor*/boards/*board*/aws_demos/config_files/ota_demo_config.h ファイルを開き、証明書を入力します。

   ```
   [] = "your-certificate-key";
   ```

3. コード署名証明書の内容を次に貼り付けます。

   ```
   #define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
   ```

   aws_clientcredential_keys.h と同じ書式に従って、各行は改行文字 ('\n') で終わり、引用符で囲む必要があります。

   たとえば、証明書は次のようになります。

   ```
   "-----BEGIN CERTIFICATE-----\n"
   "MIIBXTCCAQOgAwIBAgIJAM4DeybZcTwKMAoGCCqGSM49BAMCMCExHzAdBgNVBAMM\n"
   "FnRlc3Rf62lnbmVyQGFtYXpvbi5jb20wHhcNMTcxMTAzMTkxODM1WhcNMTgxMTAz\n"
   "MTkxODM2WjAhMR8wHQYDVQBBZZZ0ZXN0X3NpZ25lckBhbWF6b24uY29tMFkwEwYH\n"
   "KoZIzj0CAQYIKoZIzj0DAQcDQgAERavZfvwL1X+E4dIF7dbkVMUn4IrJ1CAsFkc8\n"
   "gZxPzn683H40XMKltDZPEwr9ng78w9+QYQg7ygnr2stz8yhh06MkMCIwCwYDVR0P\n"
   "BAQDAgeAMBMGA1UdJQQMMAoGCCsGAQUFBwMDMAoGCCqGSM49BAMCA0gAMEUCIF0R\n"
   "r5cb7rEUNtWOvGd05MacrgOABfSoVYvBOK9fP63WAqt5h3BaS123coKSGg84twlq\n"
   "TkO/pV/xEmyZmZdV+HxV/OM=\n"
   "-----END CERTIFICATE-----\n";
   ```

4. Python 3 以降をインストールします。
5. pip install pyopenssl を実行し、pyOpenSSL をインストールします。

6. demos/ota/bootloader/utility/codesigner_cert_utility/ パスに、コード署名証明書を .pem 形式でコピーします。証明書ファイルの名前を aws_ota_codesigner_certificate.pem に変更します。
7. *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。
8. ソリューションを構築し、エラーなしでビルドされていることを確認します。
9. ターミナルエミュレーターを起動し、次の設定を使用してボードに接続します。
   - ボーレート: 115200
   - データビット: 8
   - パリティ: なし
   - ストップビット: 1
10. ボードからデバッガーを取り外し、ボードでプロジェクトを実行して、Wi-Fi と AWS IoT MQTT メッセージブローカーに接続できることを確認します。

プロジェクトを実行すると、MPLAB X IDE は出力ウィンドウを開きます。[ICD4] タブが選択されていることを確認します。次のような出力が表示されます。

```
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
```

ターミナルエミュレーターは次のようなテキストを表示します。

```
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
```

この出力は、Microchip Curiosity PIC32MZEF が OTA 更新に必要な MQTT トピックに接続 AWS IoT してサブスクライブできることを示しています。保留中の OTA 更新ジョブがないため、`Missing job parameter` メッセージが必要です。

Espressif ESP32 での FreeRTOS OTA デモのダウンロード、構築、フラッシュ、実行

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

1. FreeRTOS のソースを GitHub からダウンロードしてください。手順については、README.md ファイルを参照してください。必要なすべてのソースとライブラリを含む IDE でプロジェクトを作成します。
2. 必要な GCC ベースのツールチェーンを設定するには、「Espressif の開始方法」の指示に従ってください。
3. *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define

CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。

4. vendors/espressif/boards/esp32/aws_demos ディレクトリで make を実行し、デモプロジェクトを構築します。Espressif の開始方法で説明されているように、デモプログラムをフラッシュし、make flash monitor を実行して出力を検証できます。
5. OTA 更新デモを実行する前に以下を確認してください。

   - *freertos*/vendors/*vendor*/boards/*board*/aws_demos/config_files/aws_demo_config.h を開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。
   - vendors/*vendor*/boards/*board*/aws_demos/config_files/ota_demo_config.h を開いて、次の場所にある SHA-256/ECDSA コード署名証明書をコピーします。

     ```
     #define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
     ```

Renesas RX65N での FreeRTOS OTA デモのダウンロード、構築、フラッシュ、および実行

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

この章では、Renesas RX65N で FreeRTOS OTA デモアプリケーションをダウンロード、構築、フラッシュ、および実行する方法について説明します。

トピック

- オペレーティング環境をセットアップする
- AWS リソースのセットアップ
- ヘッダーファイルをインポートして設定し、aws_demos と boot_loader を構築する

オペレーティング環境をセットアップする

このセクションの手順では、次の環境を使用します。

- IDE: e^2 studio 7.8.0、e^2 studio 2020-07
- ツールチェーン: CCRX Compiler v3.0.1
- ターゲットデバイス: RSKRX65N-2MB
- デバッガー: E^2、E^2 Lite エミュレータ
- ソフトウェア: Renesas Flash Programmer、Renesas Secure Flash Programmer.exe、Tera Term

ハードウェアをセットアップするには

1. E^2 Lite エミュレータと USB シリアルポートを RX65N ボードと PC に接続します。
2. 電源を RX65N に接続します。

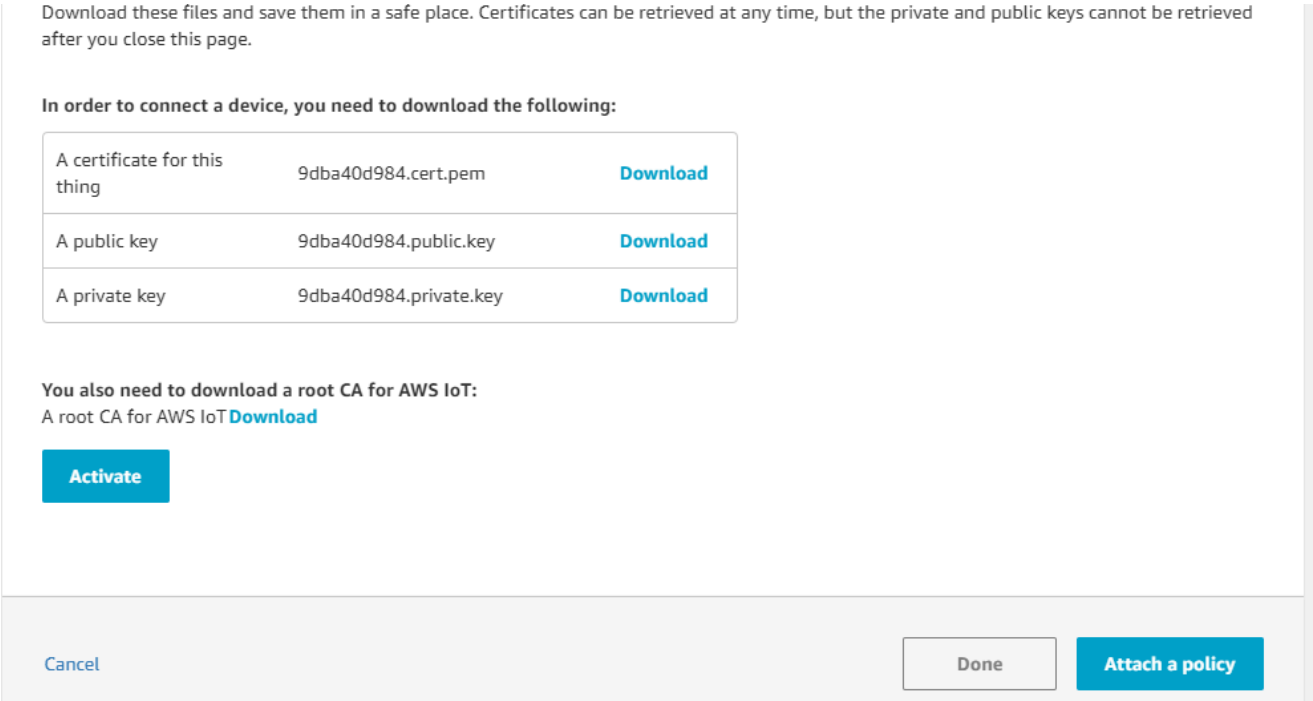
AWS リソースのセットアップ

1. FreeRTOS デモを実行するには、 AWS IoT サービスへのアクセス許可を持つ IAM ユーザーの AWS アカウントが必要です。まだの場合は、AWS アカウントとアクセス許可の設定の手順に従ってください。
2. OTA の更新を設定するには、OTA 更新の前提条件の手順に従います。特に、MQTT を使用した OTA 更新の前提条件 の手順を実行してください。
3. AWS IoT コンソールを開きます。
4. 左側のナビゲーションペインで、[Manage] (管理)、[Things] (モノ) の順に選択して、モノを作成します。

   モノとは、 AWS IoTにおける特定のデバイスまたは論理エンティティを表します。物理的なデバイスやセンサー (電球や壁のスイッチなど) は、モノとして扱うことができます。また、アプリケーションのインスタンスのような論理エンティティ、または接続しないが AWS IoT接続するデバイスに関連する物理エンティティ (エンジンセンサーやコントロールパネルがある自動車など) でもかまいません。 は、モノの管理に役立つモノレジストリ AWS IoT を提供します。

   a. [Create] (作成)、[Create a single thing] (単一のモノを作成する) の順に選択します。
   b. モノの名前を [Name] (名前) に入力し、[Next] (次へ) を選択します。
   c. [証明書の作成] を選択します。
   d. 作成された 3 つのファイルをダウンロードして、[Activate] (アクティブ化) を選択します。

e. [ポリシーのアタッチ] を選択します。



f. デバイスポリシーで作成したポリシーを選択します。

MQTT を使用して OTA 更新を受信する各デバイスは、 でモノとして登録する必要があり AWS IoT 、モノにはリストされているようなポリシーがアタッチされている必要があります。"Action" および "Resource" オブジェクトの項目の詳細については、AWS IoT Core ポリシーアクションとAWS IoT Core アクションリソースを参照してください。

メモ

- アクセス`iot:Connect`許可により、デバイスは MQTT AWS IoT 経由で に接続できます。
- AWS IoT ジョブ (`.../jobs/*`) のトピックに対する `iot:Subscribe` および `iot:Publish` アクセス許可を使用すると、接続されたデバイスがジョブ通知とジョブドキュメントを受け取り、ジョブ実行の完了状態を公開できます。
- OTA ストリーム (`.../streams/*`) AWS IoT のトピックに対する `iot:Subscribe`およびアクセス`iot:Publish`許可により、接続されたデバイスは OTA 更新データを取得できます AWS IoT。これらのアクセス許可は、MQTT を介してファームウェア更新を実行するために必要です。
- アクセス`iot:Receive`許可により、 AWS IoT Core はそれらのトピックに関するメッセージを接続されたデバイスに発行できます。このアクセス許可は、MQTT メッセージ

の配信ごとにチェックされます。このアクセス許可を使用して、トピックに現在サブスクライブしているクライアントへのアクセスを取り消すことができます。

5. コード署名プロファイルを作成し、コード署名証明書を登録するには AWS。

   a. キーと証明書を作成するには、Renesas MCU ファームウェア更新の設計ポリシーのセクション 7.3「OpenSSL で ECDSA-SHA256 キーペアを生成する」を参照してください。

   b. AWS IoT コンソールを開きます。左側のナビゲーションペインで、[Manage] (管理)、[Jobs] (ジョブ) の順に選択します。[Create a job] (ジョブの作成) を選択して、[Create OTA update Job] (OTA 更新ジョブの作成) を選択します。

   c. [Select devices to update] (更新するデバイスの選択) で、[Select] (選択) を選択して作成済みのモノを選択します。[次へ] を選択してください。

   d. [Create a FreeRTOS OTA update job] (FreeRTOS OTA 更新ジョブの作成) ページで、次の手順を実行します。

      i. [Select the protocol for firmware image transfer] (ファームウェアイメージ転送プロトコルを選択) で、[MQTT] を選択します。

      ii. [Select and sign your firmware image] (ファームウェアイメージの選択と署名) で、[Sign a new firmware image for me] (新しいファームウェアイメージに署名します) を選択します。

      iii. [Code signing profile] (コード署名プロファイル) で、[Create] (作成) を選択します。

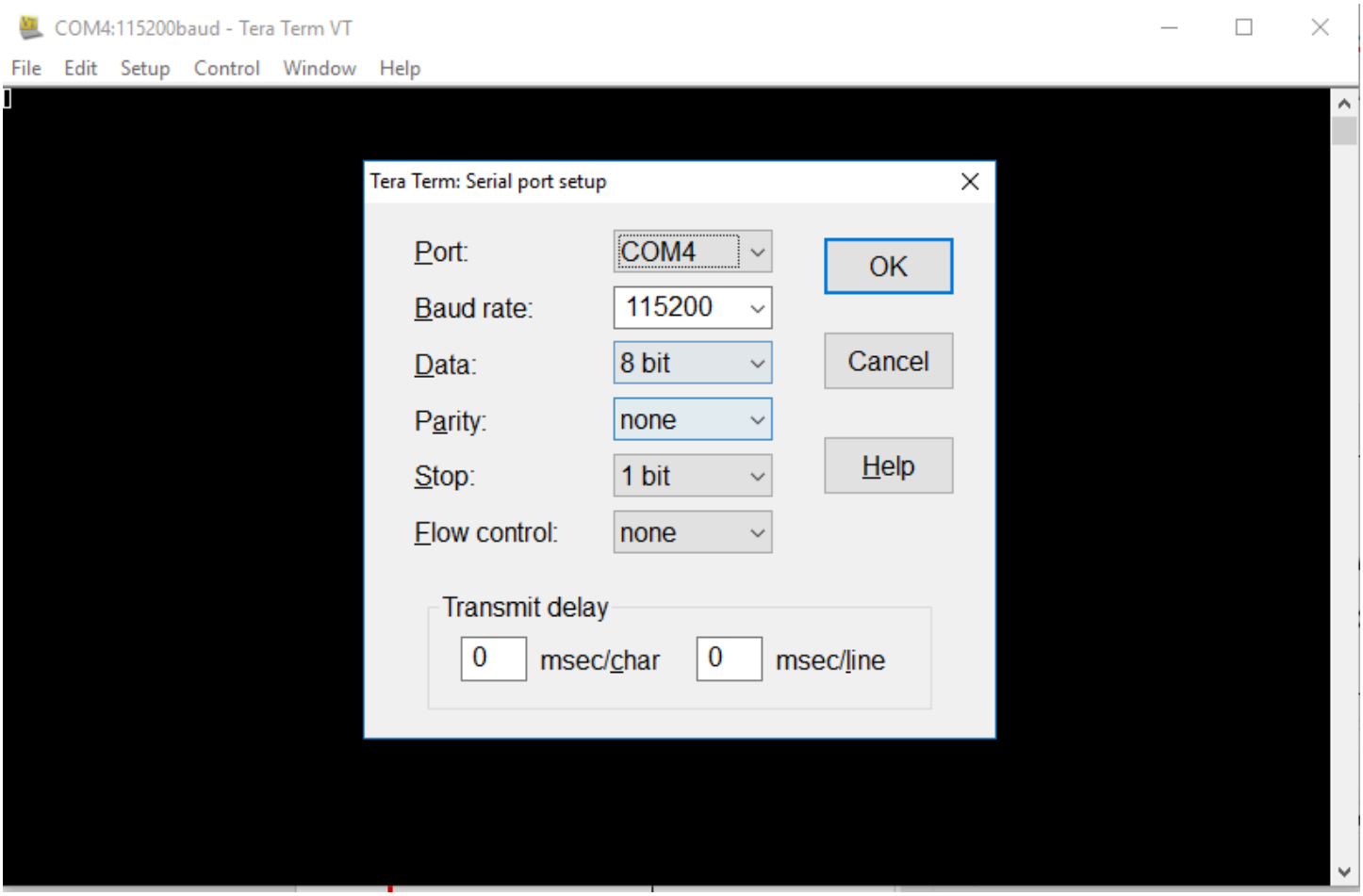
      iv. [Create a code signing profile] (コード署名プロファイルの作成) ウィンドウで、[Profile name] (プロファイル名) を入力します。[Device hardware platform] (デバイスハードウェアプラットフォーム) で、[Windows Simulator] を選択します。[Code signing certificate] (コード署名証明書) で、[Import] (インポート) を選択します。

      v. 証明書 (`secp256r1.crt`)、証明書のプライベートキー (`secp256r1.key`)、証明書チェーン (`ca.crt`) を参照して選択します。

      vi. [Pathname of code signing certificate on device] (デバイスのコード署名証明書のパス名) を入力します。次に [作成] を選択します。

6. のコード署名へのアクセスを許可するには AWS IoT、「」の手順に従いますのコード署名へのアクセスを許可する AWS IoT。

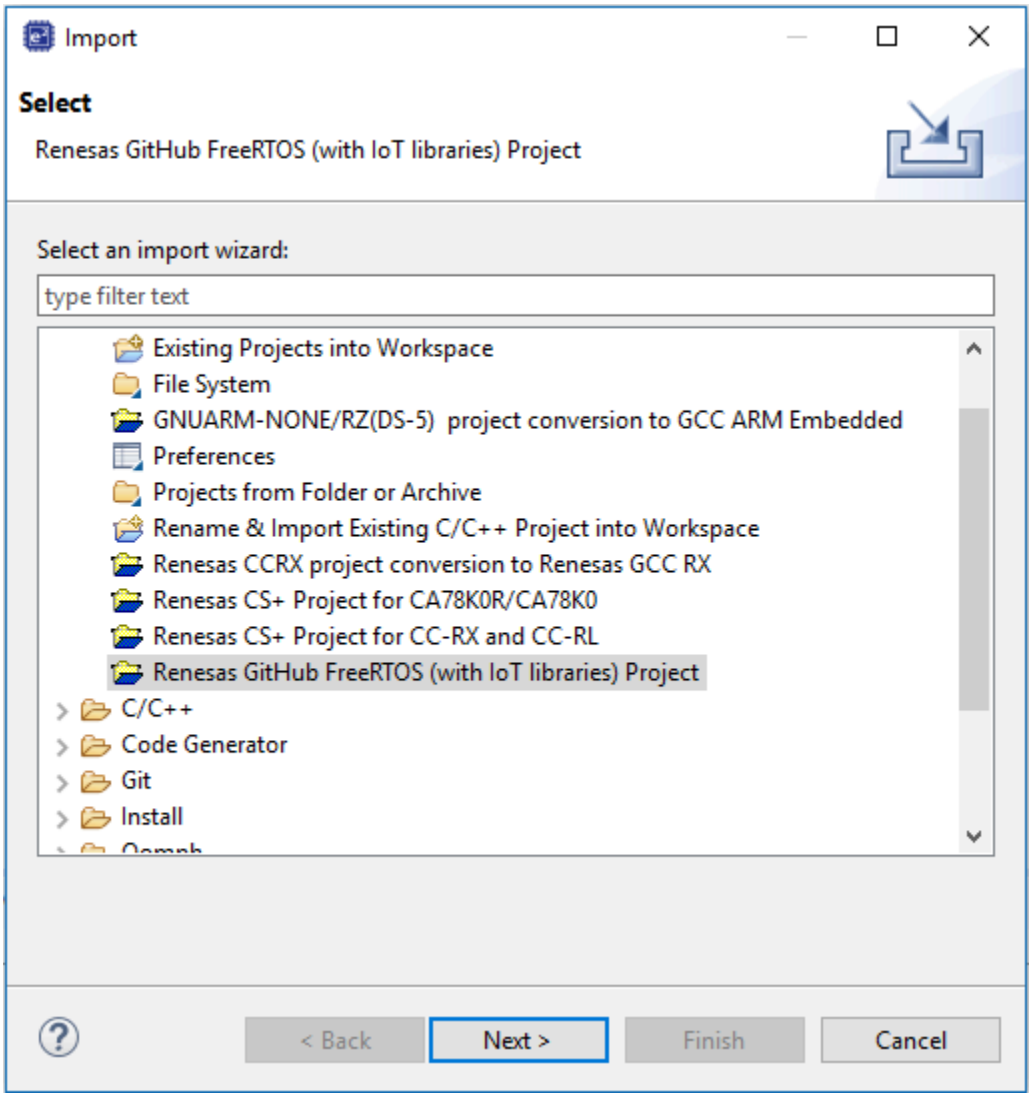
Tera Term が PC にインストールされていない場合は、以下のとおり https://ttssh2.osdn.jp/index.html.en からダウンロードしてセットアップできます。USB シリアルポートがデバイスから PC に差し込まれていることを確認します。



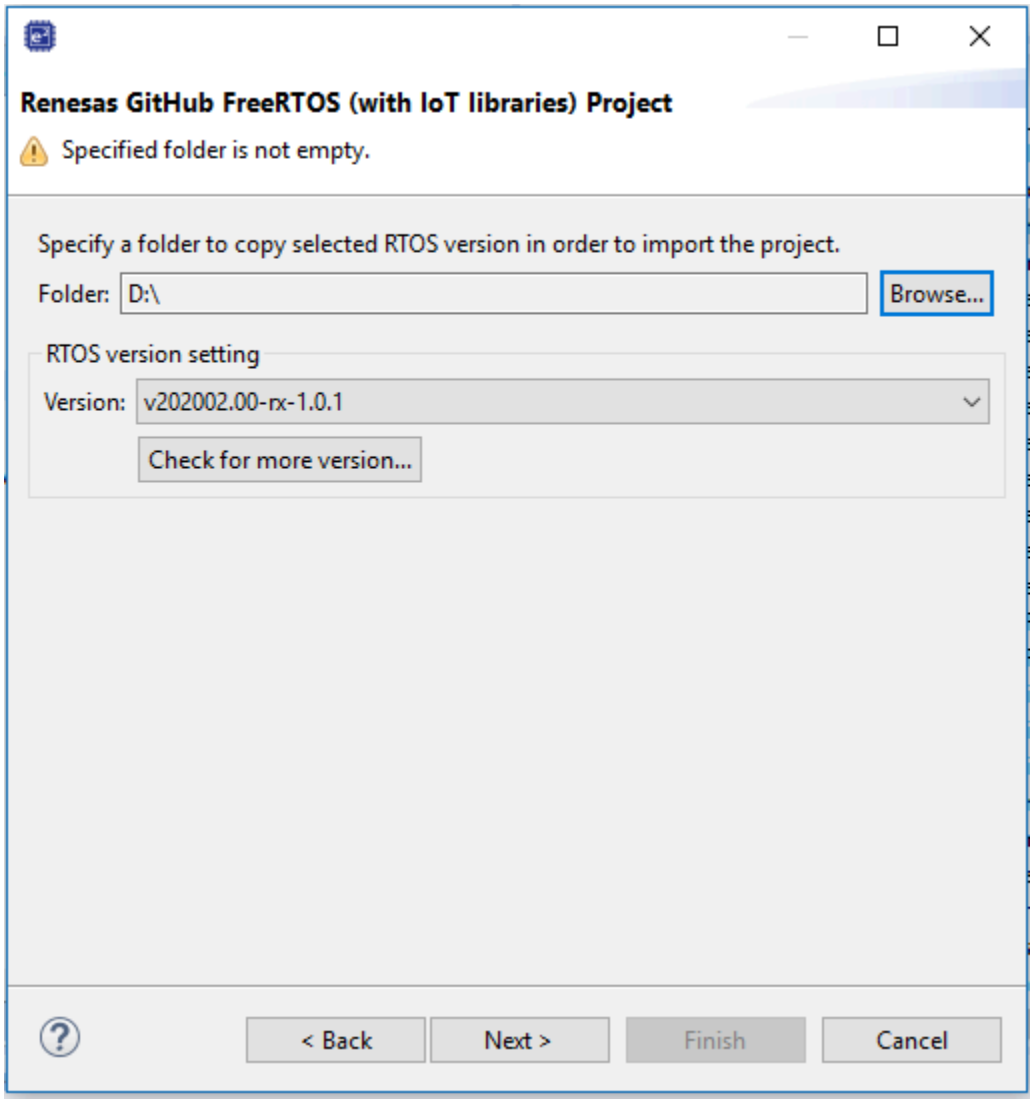
ヘッダーファイルをインポートして設定し、aws_demos と boot_loader を構築する

まず、FreeRTOS パッケージの最新バージョンを選択すると、GitHub からこれがダウンロードされ、プロジェクトに自動的にインポートされます。このようにして、FreeRTOS の設定とアプリケーションコードの記述に集中できます。

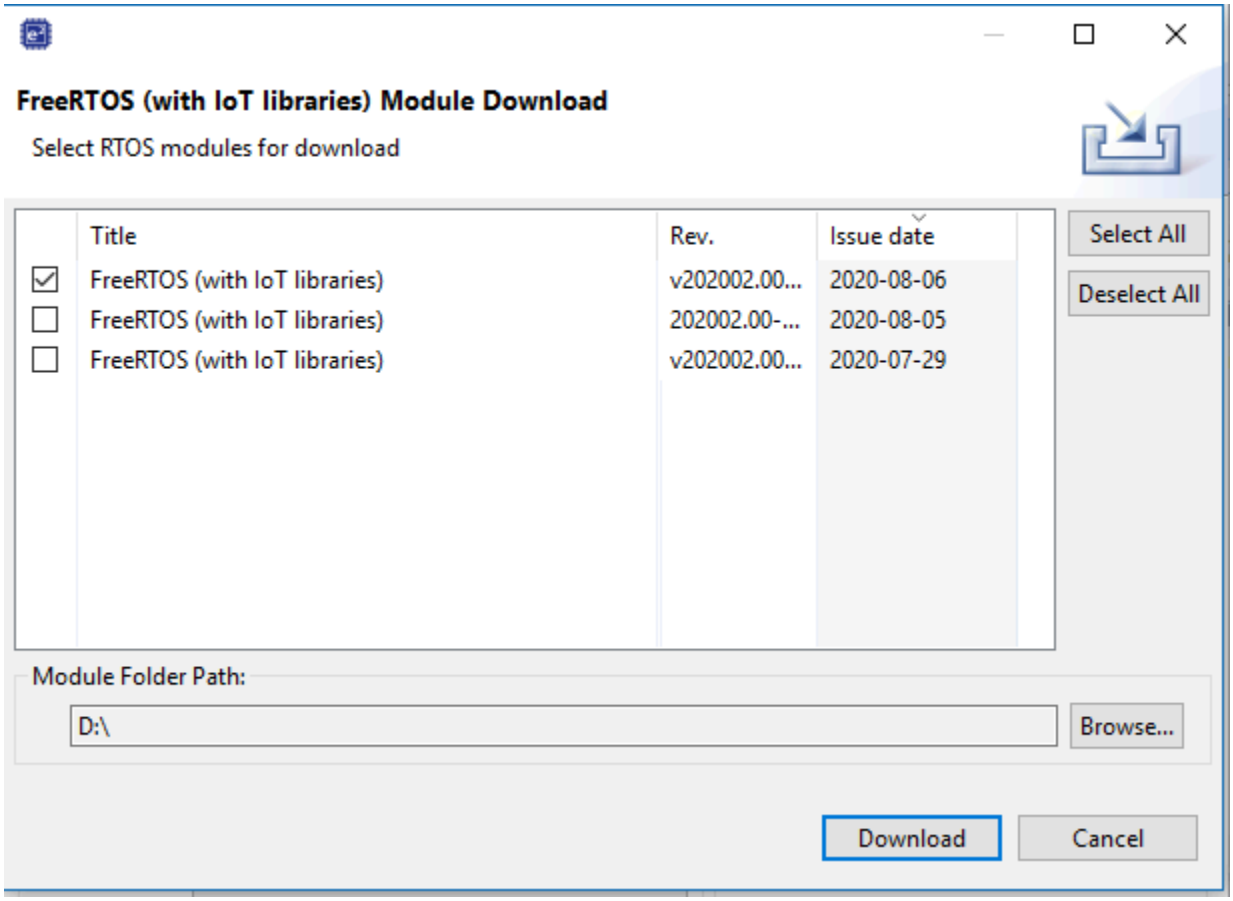
1. e^2 studio を起動します。
2. [File] (ファイル) を選択して、[Import...] (インポート...) を選択します。
3. [Renesas GitHub FreeRTOS (with IoT libraries) Project] (Renesas GitHub FreeRTOS (IoT ライブラリ) プロジェクト) を選択します。



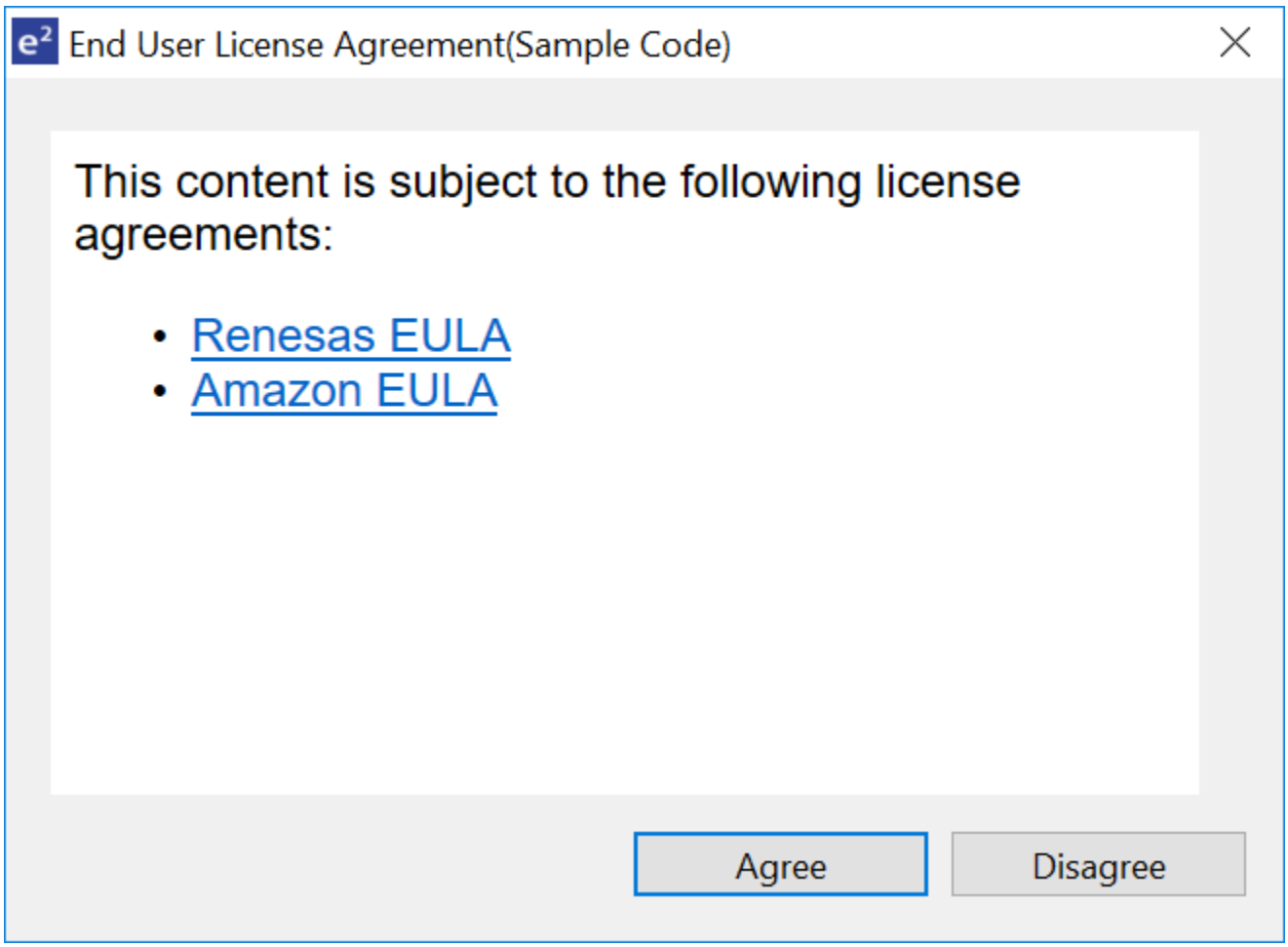
4. [Check for more version...] (他のバージョンを確認する...) を選択して、ダウンロードダイアログボックスを表示します。



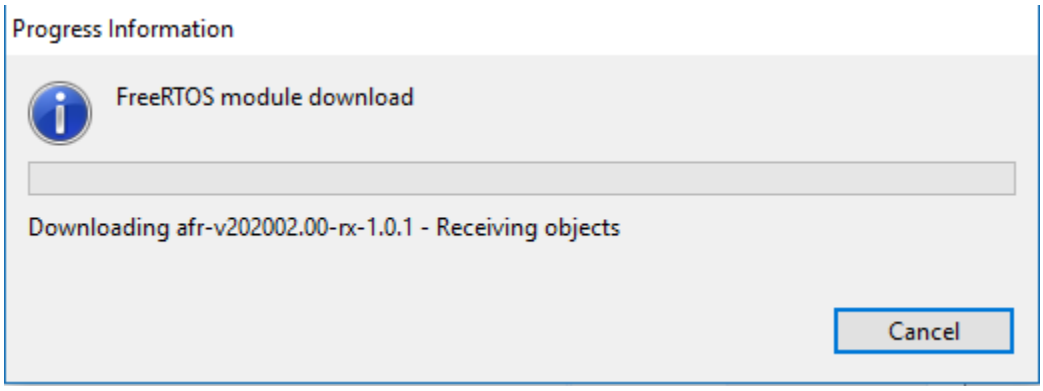
5. 最新のパッケージを選択します。



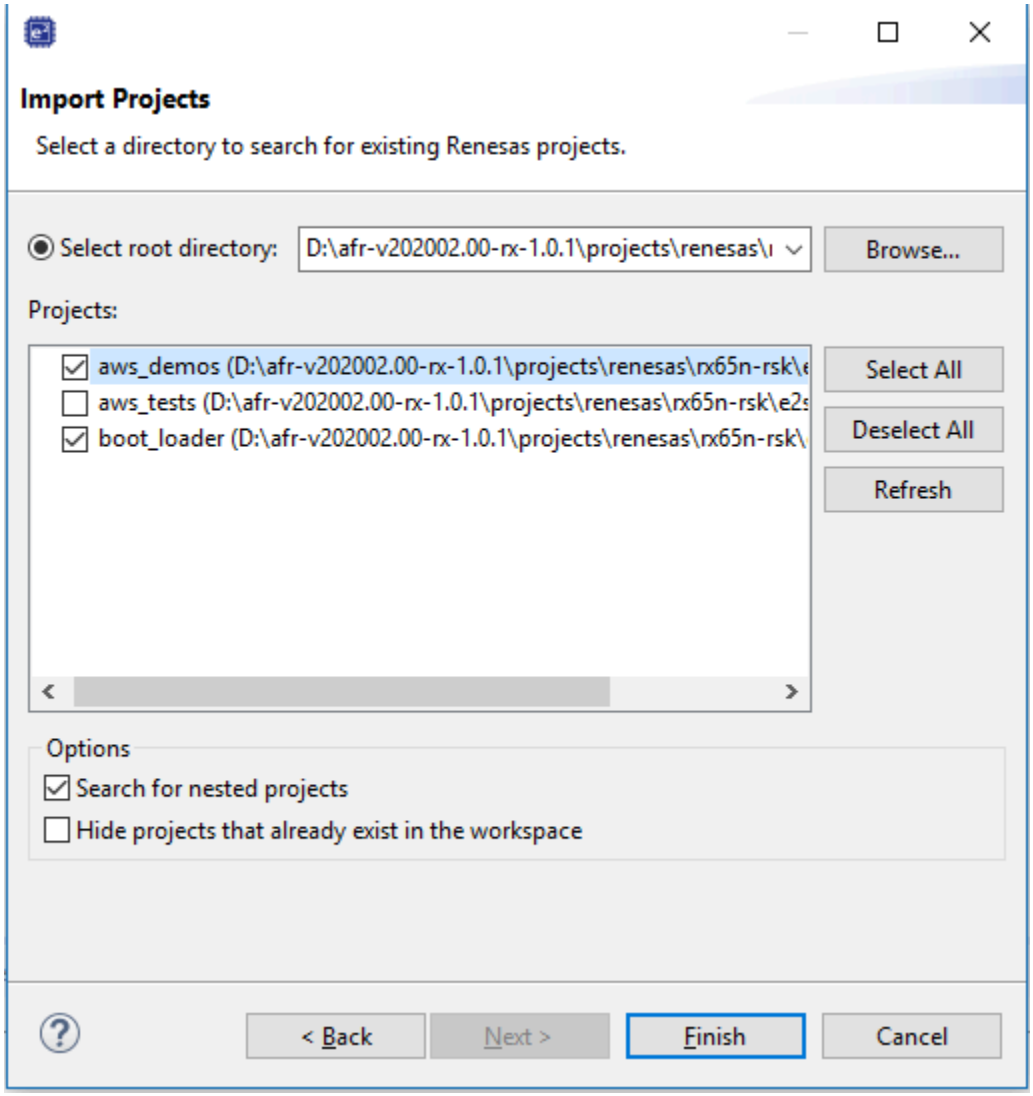
6. [Agree] (同意する) を選択して、エンドユーザー使用許諾契約に同意します。



7. ダウンロードが完了するまで待ちます。



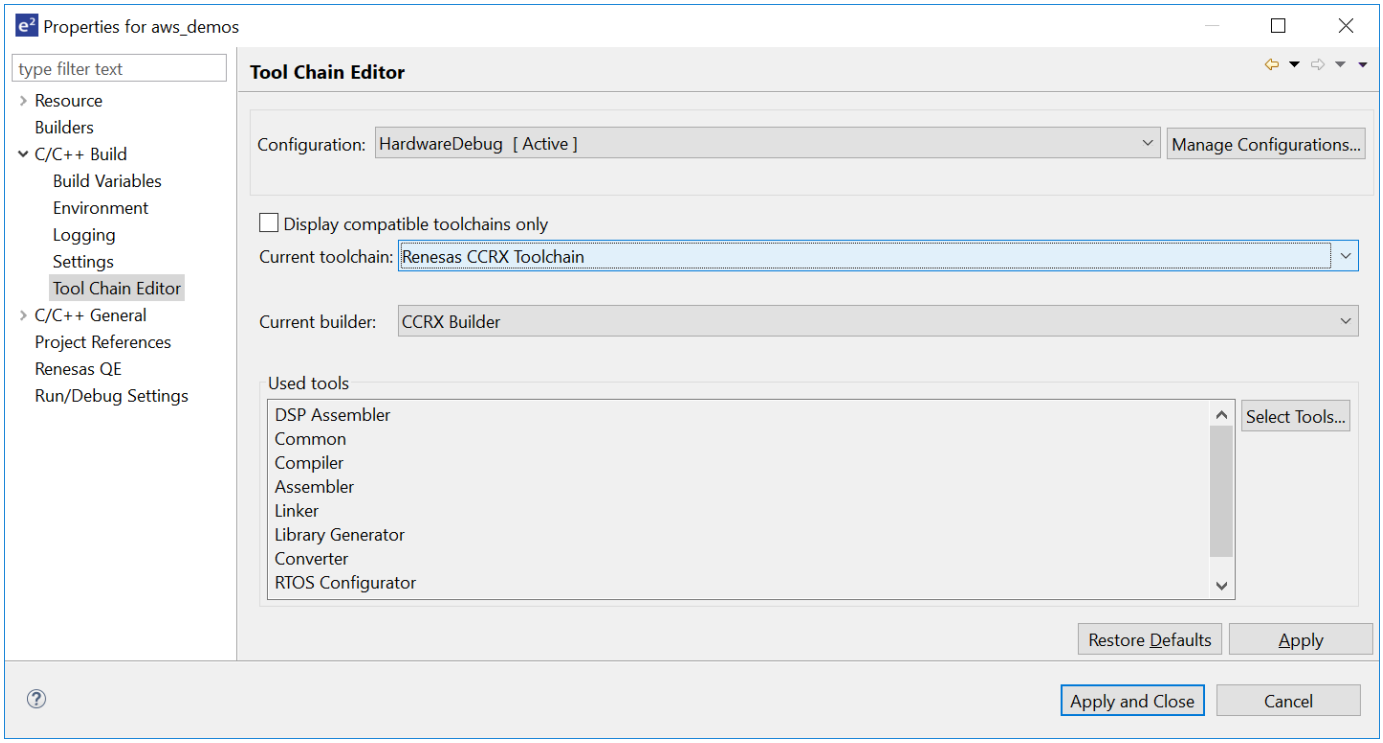
8. aws_demos と boot_loader プロジェクトを選択し、[Finish] (完了) を選択してインポートします。



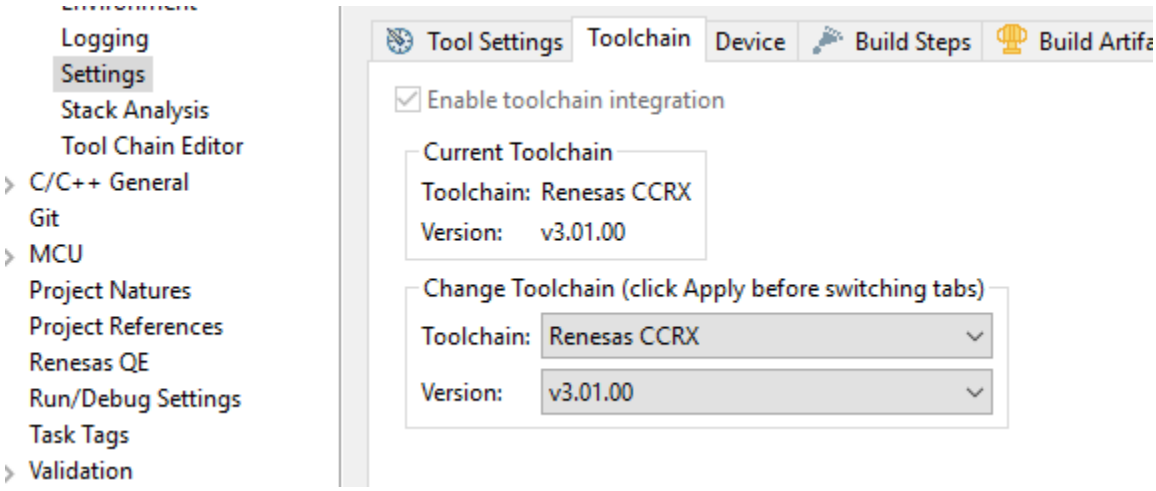
9. 両方のプロジェクトで、プロジェクトプロパティを開きます。ナビゲーションペインで [Tool Chain Editor] (ツールチェーンエディタ) を選択します。

   a. [Current toolchain] (現在のツールチェーン) を選択します。

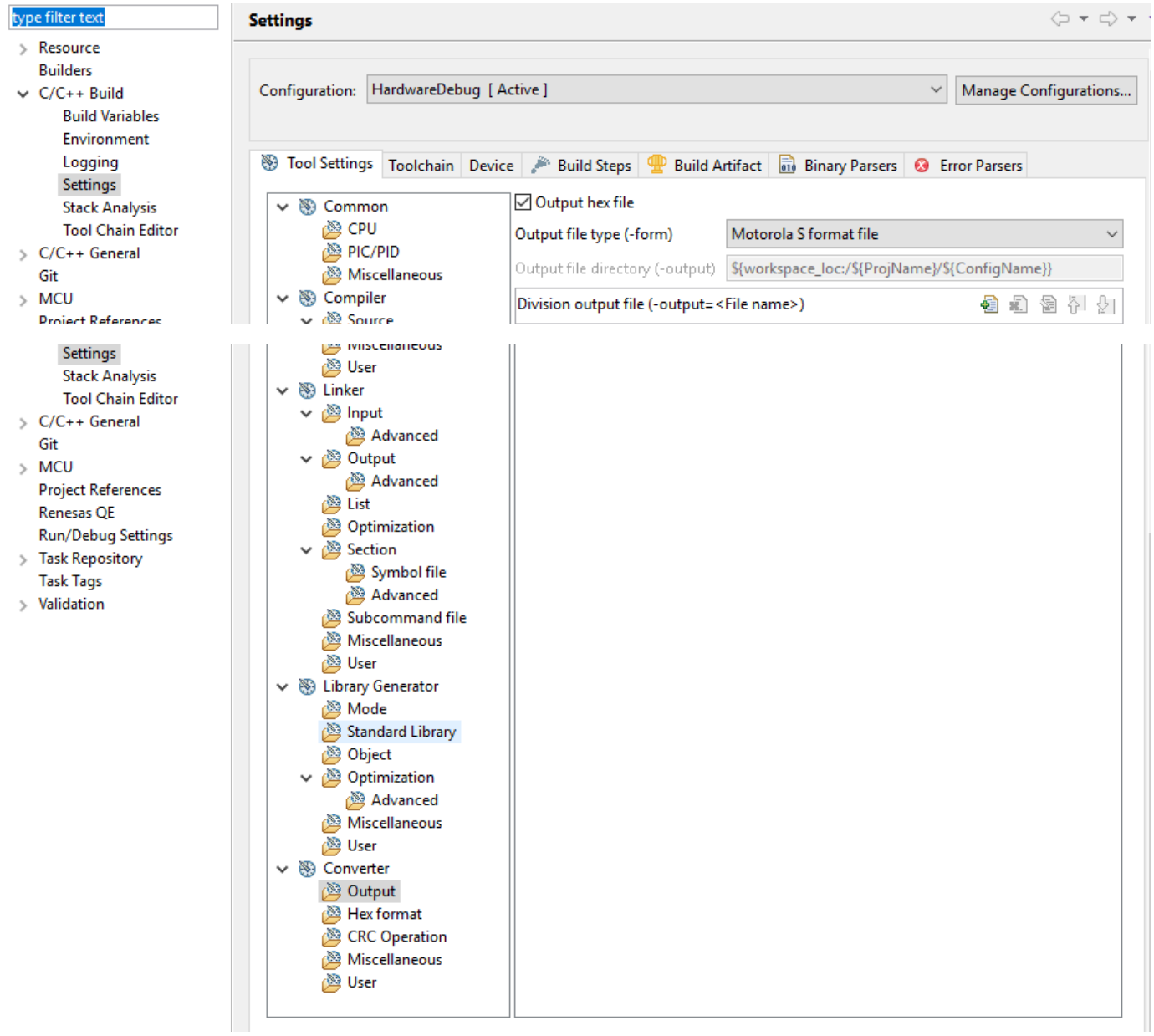
   b. [Current builder] (現在のビルダー) を選択します。



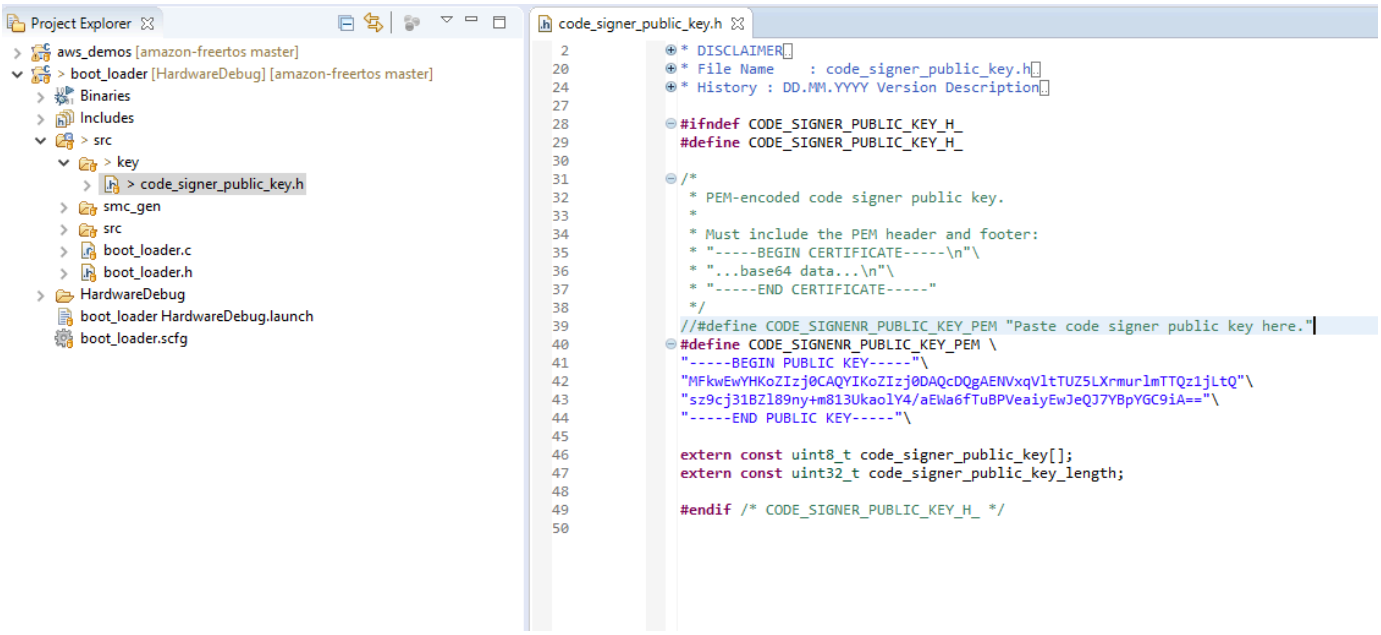
10. ナビゲーションペインで [設定] を選択します。[Toolchain] (ツールチェーン) タブで、ツールチェーンの [Version] (バージョン) を選択します。



[Tool Settings] (ツール設定) タブを選択して、[Converter] (コンバーター) を展開し、[Output] (出力) を選択します。メインウィンドウで [Output hex file] (16 進数ファイルの出力) が選択されていることを確認し、[Output file type] (出力ファイルタイプ) を選びます。



11. bootloader プロジェクトで、`projects\renesas\rx65n-rsk\e2studio\boot_loader\src\key\code_signer_public_key.h` を開き、パブリックキーを入力します。パブリックキーの作成方法については、Amazon Web Services を使用して RX65N で FreeRTOS OTA を実装する方法と Renesas MCU ファームウェア更新の設計ポリシーのセクション 7.3「OpenSSL で ECDSA-SHA256 キーペアを生成する」を参照してください。



次にプロジェクトを構築して、boot_loader.mot を作成します。

12. aws_demos プロジェクトを開きます。

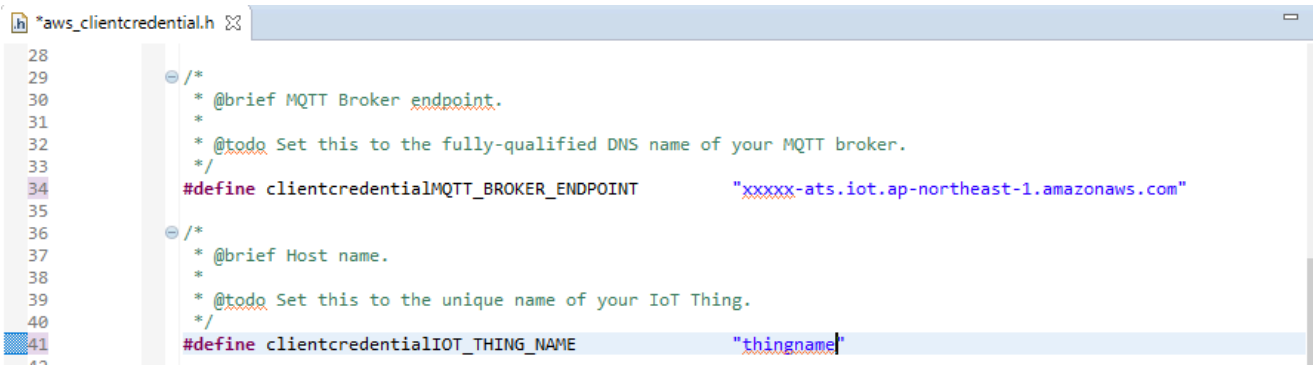
    a. AWS IoT コンソールを開きます。

    b. 左のナビゲーションペインの [Settings] (設定) を選択します。[Device data endpoint] (デバイスデータエンドポイント) テキストボックスのカスタムエンドポイントをメモします。

    c. [Manage] (管理) を選択して、[Things] (モノ) を選択します。ボードの AWS IoT モノの名前を書き留めます。

    d. aws_demos プロジェクトで、demos/include/aws_clientcredential.h を開き、次の値を指定します。

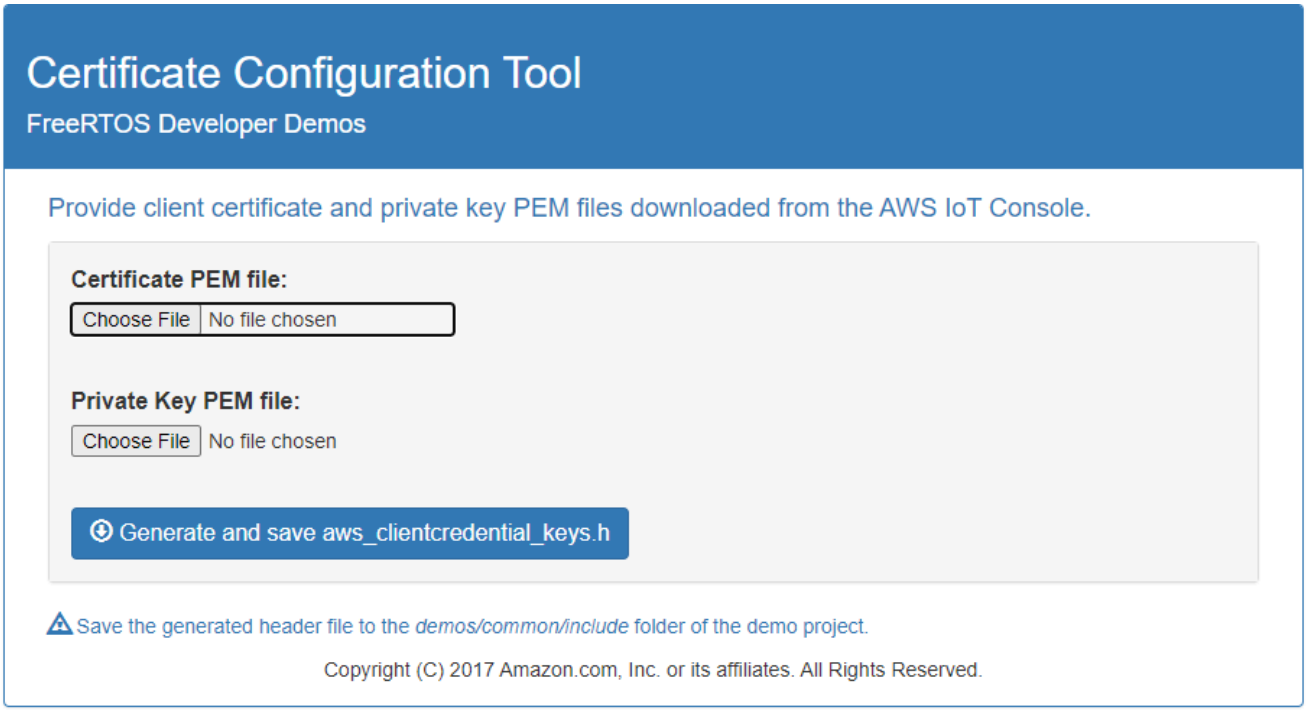
```
#define clientcredentialMQTT_BROKER_ENDPOINT[] = "Your AWS IoT endpoint";
#define clientcredentialIOT_THING_NAME "The AWS IoT thing name of your board"
```



e. `tools/certificate_configuration/CertificateConfigurator.html` ファイルを開きます。

f. ダウンロード済みの証明書 PEM ファイルとプライベートキー PEM ファイルをインポートします。

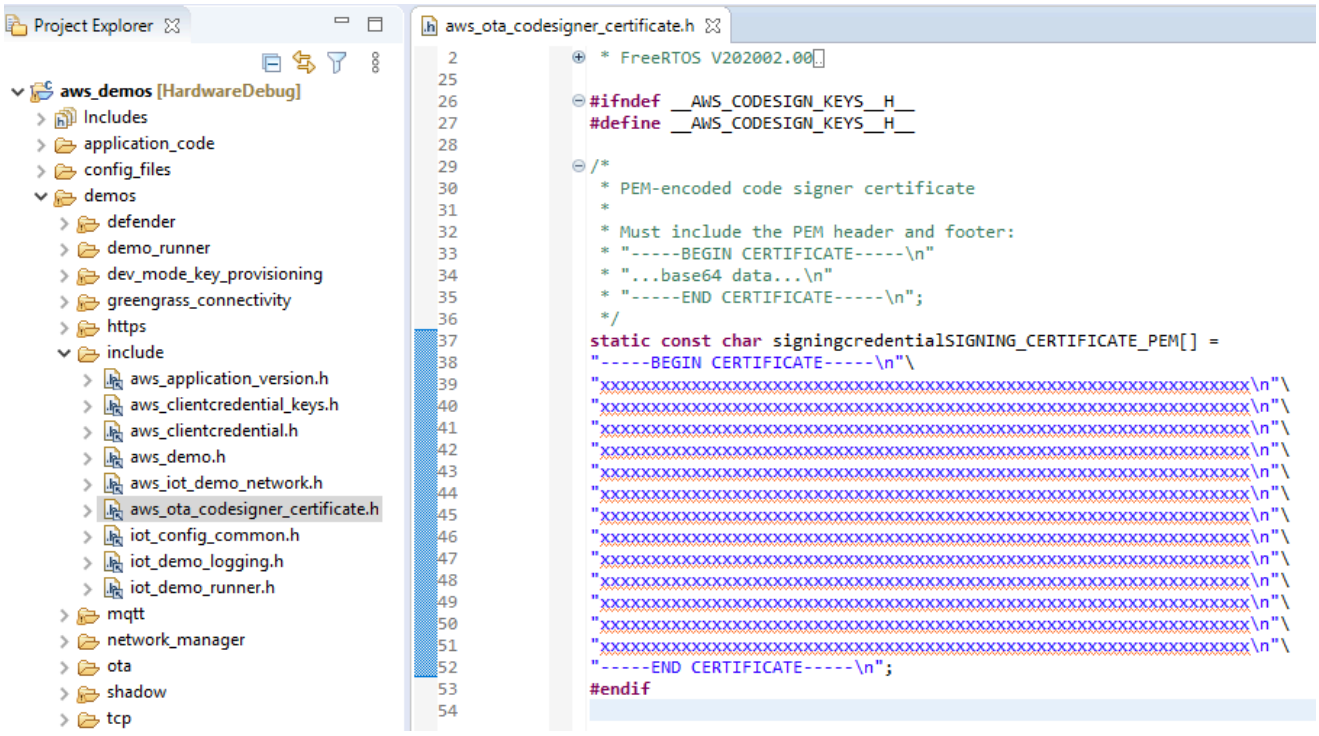
g. [Generate and save aws_clientcredential_keys.h] (aws_clientcredential_keys.h の生成と保存) を選択して、`demos/include/` ディレクトリのこのファイルを置き換えます。



h. `vendors/renesas/boards/rx65n-rsk/aws_demos/config_files/ota_demo_config.h` ファイルを開いて、次の値を指定します。

```
#define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
```

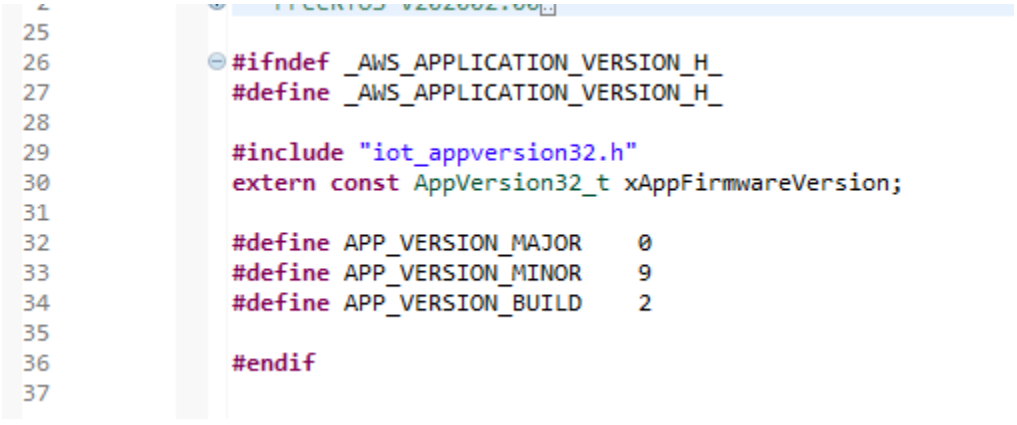
ここで、*`your-certificate-key`* はファイル `secp256r1.crt` の値です。証明書の各行の後に必ず「\」を追加してください。`secp256r1.crt` ファイルの作成方法については、Amazon Web Services を使用して RX65N で FreeRTOS OTA を実装する方法と Renesas MCU ファームウェア更新の設計ポリシーのセクション 7.3「OpenSSL で ECDSA-SHA256 キーペアを生成する」を参照してください。



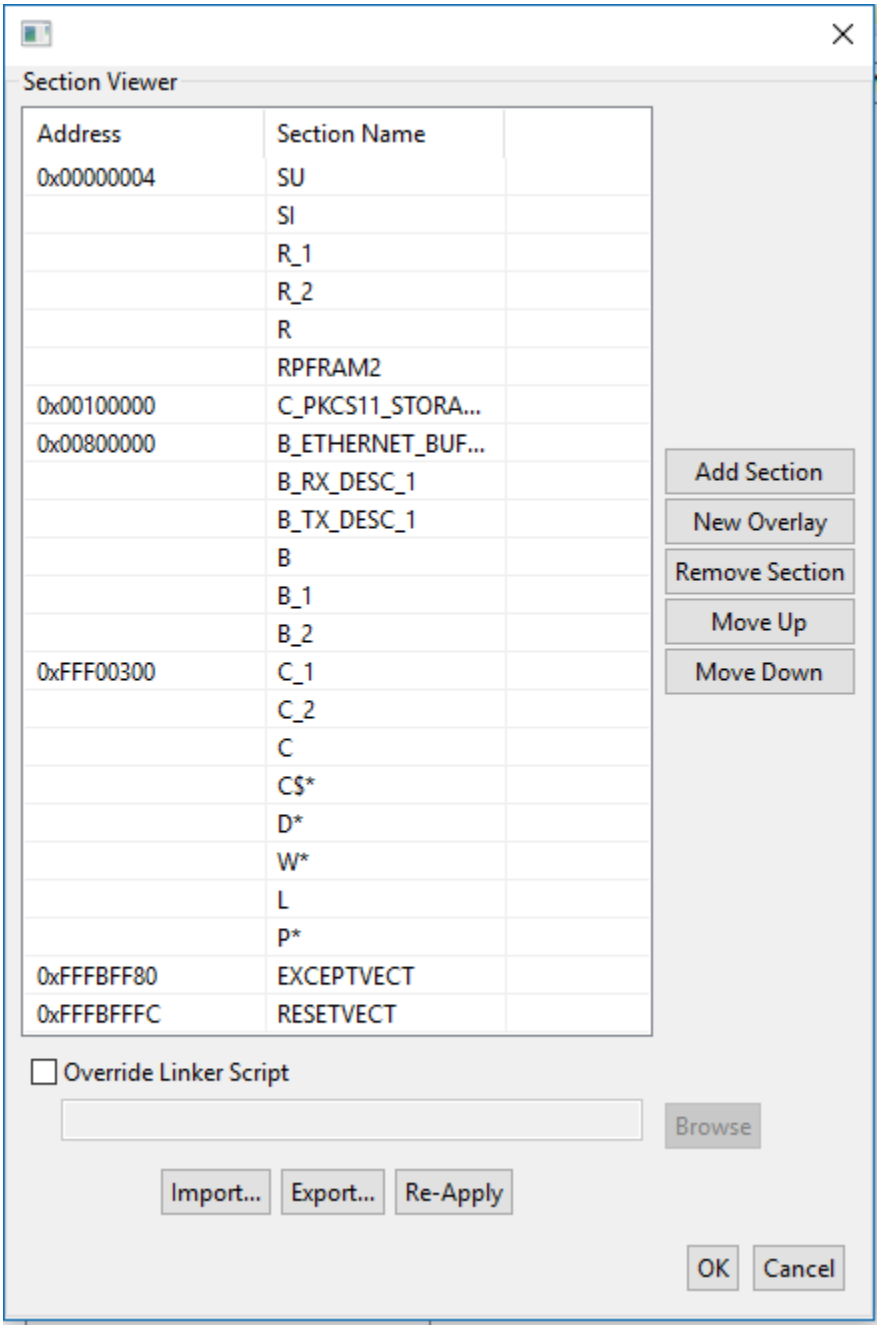
13. タスク A: ファームウェアの初期バージョンをインストールする

    a. vendors/renesas/boards/board/aws_demos/config_files/aws_demo_config.h ファイルを開き、#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED をコメントアウトして、CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。

    b. demos/include/ aws_application_version.h ファイルを開き、ファームウェアの初期バージョンを 0.9.2 に設定します。



    c. [Section Viewer] (セクションビューワー) で以下の設定を変更します。



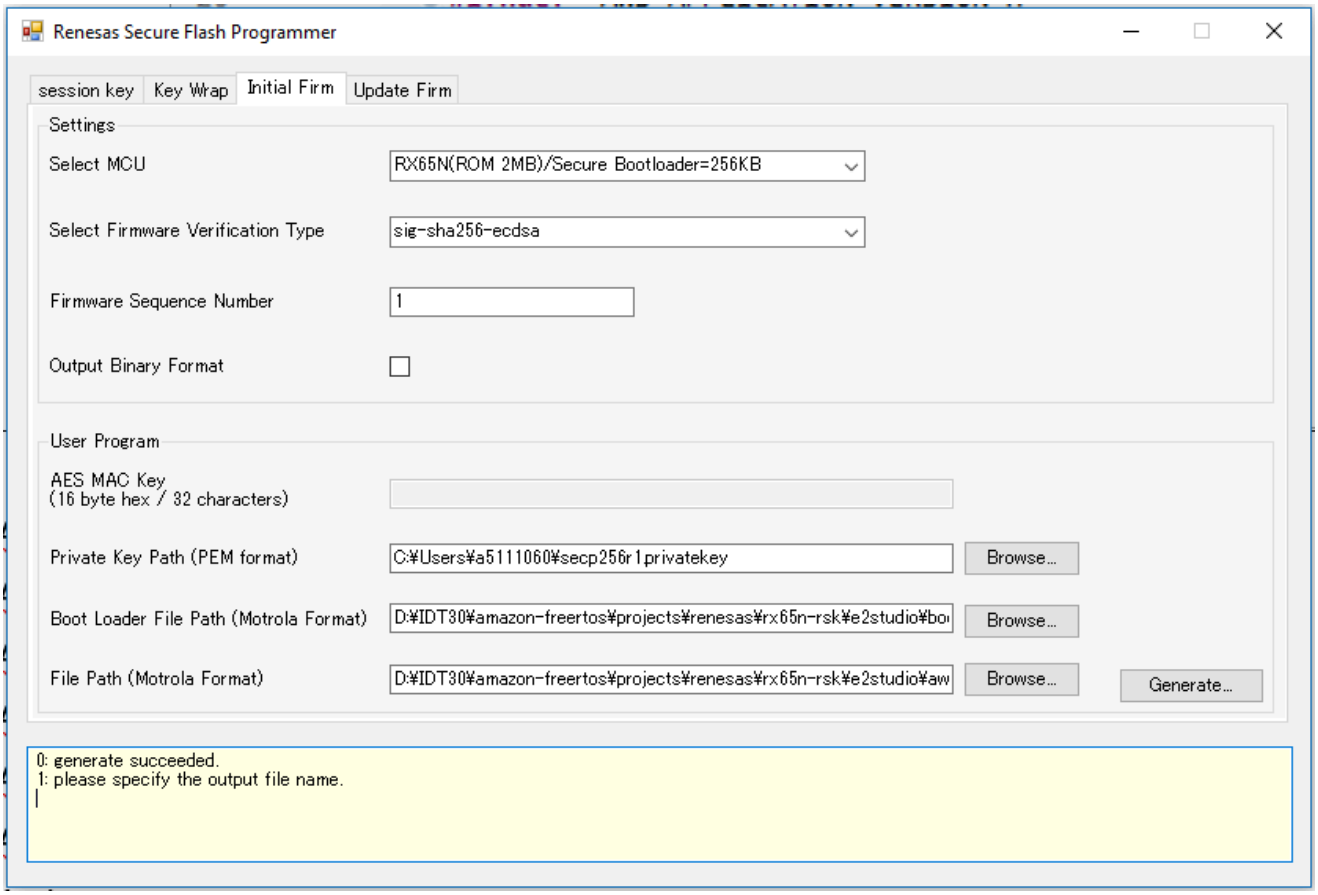
d. [Build] (構築) を選択して、`aws_demos.mot` ファイルを作成します。

14. Renesas Secure Flash Programmer を使用してファイル `userprog.mot` を作成します。`userprog.mot` は `aws_demos.mot` と `boot_loader.mot` の組み合わせです。このファイルを RX65N-RSK にフラッシュすると、初期ファームウェアをインストールできます。

    a. https://github.com/renesas/Amazon-FreeRTOS-Tools をダウンロードして、`Renesas Secure Flash Programmer.exe` を開きます。

    b. [Initial Firm] (初期ファーム) タブを選択し、以下のパラメータを設定します。

- [Private Key Path] (プライベートキーのパス): `secp256r1.privatekey` の場所。
- [Boot Loader File Path] (ブートローダーファイルパス): `boot_loader.mot` の場所 (`projects\renesas\rx65n-rsk\e2studio\boot_loader\HardwareDebug`)。
- [File Path] (ファイルパス): `aws_demos.mot` の場所 (`projects\renesas\rx65n-rsk\e2studio\aws_demos\HardwareDebug`)。



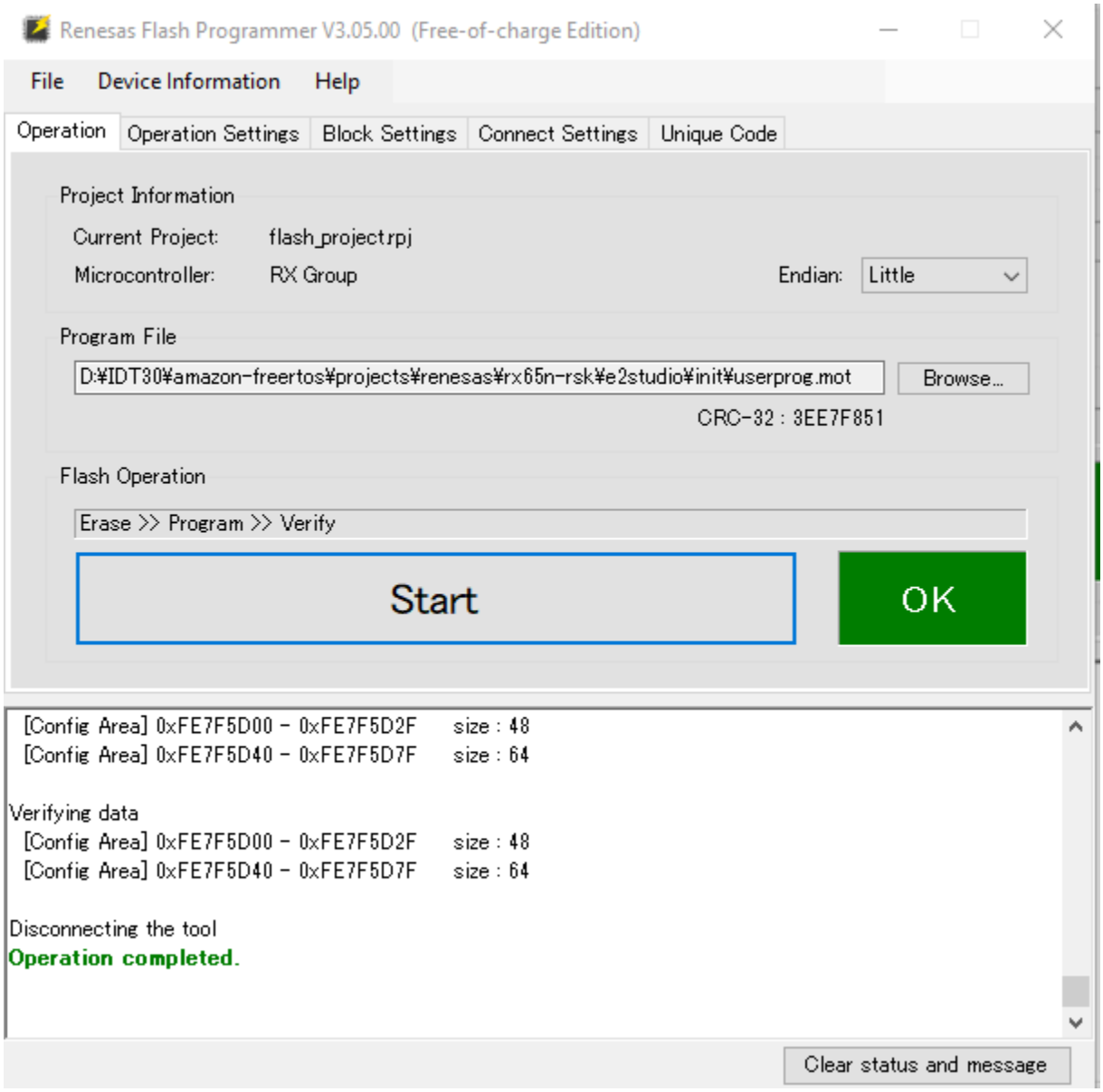
c. `init_firmware` という名前のディレクトリを作成し、`userprog.mot` を生成して `init_firmware` ディレクトリに保存します。生成が成功したことを確認します。

15. RX65N-RSK で初期ファームウェアをフラッシュします。

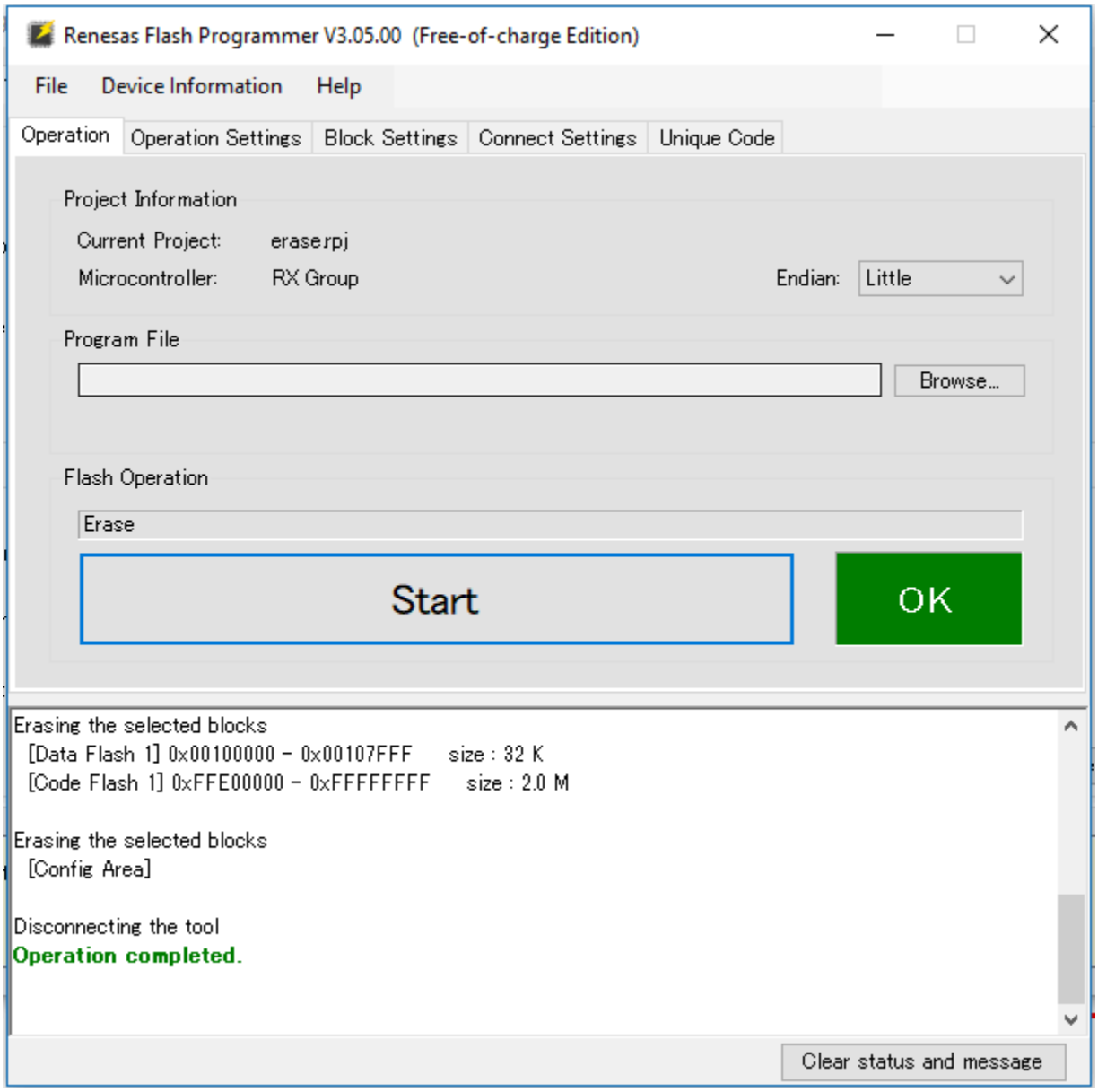
a. Renesas Flash Programmer (プログラミング GUI) の最新バージョンを https://www.renesas.com/tw/en/products/software-tools/tools/programmer/renesas-flash-programmer-programming-gui.html からダウンロードします。

b. `vendors\renesas\rx_mcu_boards\boards\rx65n-rsk\aws_demos\flash_project\erase_from_bank\ erase.rpj` ファイルを開いて銀行のデータを消去します。

c. [Start] (開始) を選択して、銀行を消去します。



d. `userprog.mot` をフラッシュするには、[Browse...] (参照...) を選択して `init_firmware` ディレクトリに移動し、`userprog.mot` ファイルを選択して、[Start] (開始) を選択します。



16. バージョン 0.9.2 (初期バージョン) のファームウェアが RX65N-RSK にインストールされました。RX65N-RSK ボードは OTA 更新をリッスンするようになりました。PC で Tera Term を開いている場合、初期ファームウェアの実行時に次のように表示されます。

```
------------------------------------------------
RX65N secure boot program
------------------------------------------------
Checking flash ROM status.
bank 0 status = 0xff [LIFECYCLE_STATE_BLANK]
bank 1 status = 0xfc [LIFECYCLE_STATE_INSTALLING]
bank info = 1. (start bank = 0)
start installing user program.
```

```
copy secure boot (part1) from bank0 to bank1...OK
copy secure boot (part2) from bank0 to bank1...OK
update LIFECYCLE_STATE from [LIFECYCLE_STATE_INSTALLING] to [LIFECYCLE_STATE_VALID]
bank1(temporary area) block0 erase (to update LIFECYCLE_STATE)...OK
bank1(temporary area) block0 write (to update LIFECYCLE_STATE)...OK
swap bank...
-----------------------------------------------
RX65N secure boot program
-----------------------------------------------
Checking flash ROM status.
bank 0 status = 0xf8 [LIFECYCLE_STATE_VALID]
bank 1 status = 0xff [LIFECYCLE_STATE_BLANK]
bank info = 0. (start bank = 1)
integrity check scheme = sig-sha256-ecdsa
bank0(execute area) on code flash integrity check...OK
jump to user program
#0 1 [ETHER_RECEI] Deferred Interrupt Handler Task started
1 1 [ETHER_RECEI] Network buffers: 3 lowest 3
2 1 [ETHER_RECEI] Heap: current 234192 lowest 234192
3 1 [ETHER_RECEI] Queue space: lowest 8
4 1 [IP-task] InitializeNetwork returns OK
5 1 [IP-task] xNetworkInterfaceInitialise returns 0
6 101 [ETHER_RECEI] Heap: current 234592 lowest 233392
7 2102 [ETHER_RECEI] prvEMACHandlerTask: PHY LS now 1
8 3001 [IP-task] xNetworkInterfaceInitialise returns 1
9 3092 [ETHER_RECEI] Network buffers: 2 lowest 2
10 3092 [ETHER_RECEI] Queue space: lowest 7
11 3092 [ETHER_RECEI] Heap: current 233320 lowest 233320
12 3193 [ETHER_RECEI] Heap: current 233816 lowest 233120
13 3593 [IP-task] vDHCPProcess: offer c0a80a09ip
14 3597 [ETHER_RECEI] Heap: current 233200 lowest 233000
15 3597 [IP-task] vDHCPProcess: offer c0a80a09ip
16 3597 [IP-task] IP Address: 192.168.10.9
17 3597 [IP-task] Subnet Mask: 255.255.255.0
18 3597 [IP-task] Gateway Address: 192.168.10.1
19 3597 [IP-task] DNS Server Address: 192.168.10.1
20 3600 [Tmr Svc] The network is up and running
21 3622 [Tmr Svc] Write certificate...
22 3697 [ETHER_RECEI] Heap: current 232320 lowest 230904
23 4497 [ETHER_RECEI] Heap: current 226344 lowest 225944
24 5317 [iot_thread] [INFO ][DEMO][5317] ---------STARTING DEMO---------

25 5317 [iot_thread] [INFO ][INIT][5317] SDK successfully initialized.
```

```
26 5317 [iot_thread] [INFO ][DEMO][5317] Successfully initialized the demo. Network
 type for the demo: 4
27 5317 [iot_thread] [INFO ][MQTT][5317] MQTT library successfully initialized.
28 5317 [iot_thread] [INFO ][DEMO][5317] OTA demo version 0.9.2

29 5317 [iot_thread] [INFO ][DEMO][5317] Connecting to broker...

30 5317 [iot_thread] [INFO ][DEMO][5317] MQTT demo client identifier is rx65n-gr-
rose (length 13).
31 5325 [ETHER_RECEI] Heap: current 206944 lowest 206504
32 5325 [ETHER_RECEI] Heap: current 206440 lowest 206440
33 5325 [ETHER_RECEI] Heap: current 206240 lowest 206240
38 5334 [ETHER_RECEI] Heap: current 190288 lowest 190288
39 5334 [ETHER_RECEI] Heap: current 190088 lowest 190088
40 5361 [ETHER_RECEI] Heap: current 158512 lowest 158168
41 5363 [ETHER_RECEI] Heap: current 158032 lowest 158032
42 5364 [ETHER_RECEI] Network buffers: 1 lowest 1
43 5364 [ETHER_RECEI] Heap: current 156856 lowest 156856
44 5364 [ETHER_RECEI] Heap: current 156656 lowest 156656
46 5374 [ETHER_RECEI] Heap: current 153016 lowest 152040
47 5492 [ETHER_RECEI] Heap: current 141464 lowest 139016
48 5751 [ETHER_RECEI] Heap: current 140160 lowest 138680
49 5917 [ETHER_RECEI] Heap: current 138280 lowest 138168
59 7361 [iot_thread] [INFO ][MQTT][7361] Establishing new MQTT connection.
62 7428 [iot_thread] [INFO ][MQTT][7428] (MQTT connection 81cfc8, CONNECT operation
 81d0e8) Wait complete with result SUCCESS.
63 7428 [iot_thread] [INFO ][MQTT][7428] New MQTT connection 4e8c established.
64 7430 [iot_thread] [OTA_AgentInit_internal] OTA Task is Ready.
65 7430 [OTA Agent T] [prvOTAAgentTask] Called handler. Current State [Ready] Event
 [Start] New state [RequestingJob]
66 7431 [OTA Agent T] [INFO ][MQTT][7431] (MQTT connection 81cfc8) SUBSCRIBE
 operation scheduled.
67 7431 [OTA Agent T] [INFO ][MQTT][7431] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Waiting for operation completion.
68 7436 [ETHER_RECEI] Heap: current 128248 lowest 127992
69 7480 [OTA Agent T] [INFO ][MQTT][7480] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Wait complete with result SUCCESS.
70 7480 [OTA Agent T] [prvSubscribeToJobNotificationTopics] OK: $aws/things/rx65n-
gr-rose/jobs/$next/get/accepted
71 7481 [OTA Agent T] [INFO ][MQTT][7481] (MQTT connection 81cfc8) SUBSCRIBE
 operation scheduled.
72 7481 [OTA Agent T] [INFO ][MQTT][7481] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Waiting for operation completion.
```

```
73 7530 [OTA Agent T] [INFO ][MQTT][7530] (MQTT connection 81cfc8, SUBSCRIBE
 operation 818c48) Wait complete with result SUCCESS.
74 7530 [OTA Agent T] [prvSubscribeToJobNotificationTopics] OK: $aws/things/rx65n-
gr-rose/jobs/notify-next
75 7530 [OTA Agent T] [prvRequestJob_Mqtt] Request #0
76 7532 [OTA Agent T] [INFO ][MQTT][7532] (MQTT connection 81cfc8) MQTT PUBLISH
 operation queued.
77 7532 [OTA Agent T] [INFO ][MQTT][7532] (MQTT connection 81cfc8, PUBLISH
 operation 818b80) Waiting for operation completion.
78 7552 [OTA Agent T] [INFO ][MQTT][7552] (MQTT connection 81cfc8, PUBLISH
 operation 818b80) Wait complete with result SUCCESS.
79 7552 [OTA Agent T] [prvOTAAgentTask] Called handler. Current State
 [RequestingJob] Event [RequestJobDocument] New state [WaitingForJob]
80 7552 [OTA Agent T] [prvParseJSONbyModel] Extracted parameter [ clientToken:
 0:rx65n-gr-rose ]
81 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: execution
82 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: jobId
83 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: jobDocument
84 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: afr_ota
85 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: protocols
86 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: files
87 7552 [OTA Agent T] [prvParseJSONbyModel] parameter not present: filepath
99 7651 [ETHER_RECEI] Heap: current 129720 lowest 127304
100 8430 [iot_thread] [INFO ][DEMO][8430] State: Ready  Received: 1   Queued: 0
 Processed: 0   Dropped: 0
101 9430 [iot_thread] [INFO ][DEMO][9430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
102 10430 [iot_thread] [INFO ][DEMO][10430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
103 11430 [iot_thread] [INFO ][DEMO][11430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
104 12430 [iot_thread] [INFO ][DEMO][12430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
105 13430 [iot_thread] [INFO ][DEMO][13430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
106 14430 [iot_thread] [INFO ][DEMO][14430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
107 15430 [iot_thread] [INFO ][DEMO][15430] State: WaitingForJob  Received: 1
 Queued: 0   Processed: 0   Dropped: 0
```

17. タスク B: ファームウェアのバージョンを更新する

    a. demos/include/aws_application_version.h ファイルを開き、APP_VERSION_BUILD トークン値を 0.9.3 に増やします。

b. プロジェクトを再構築します。

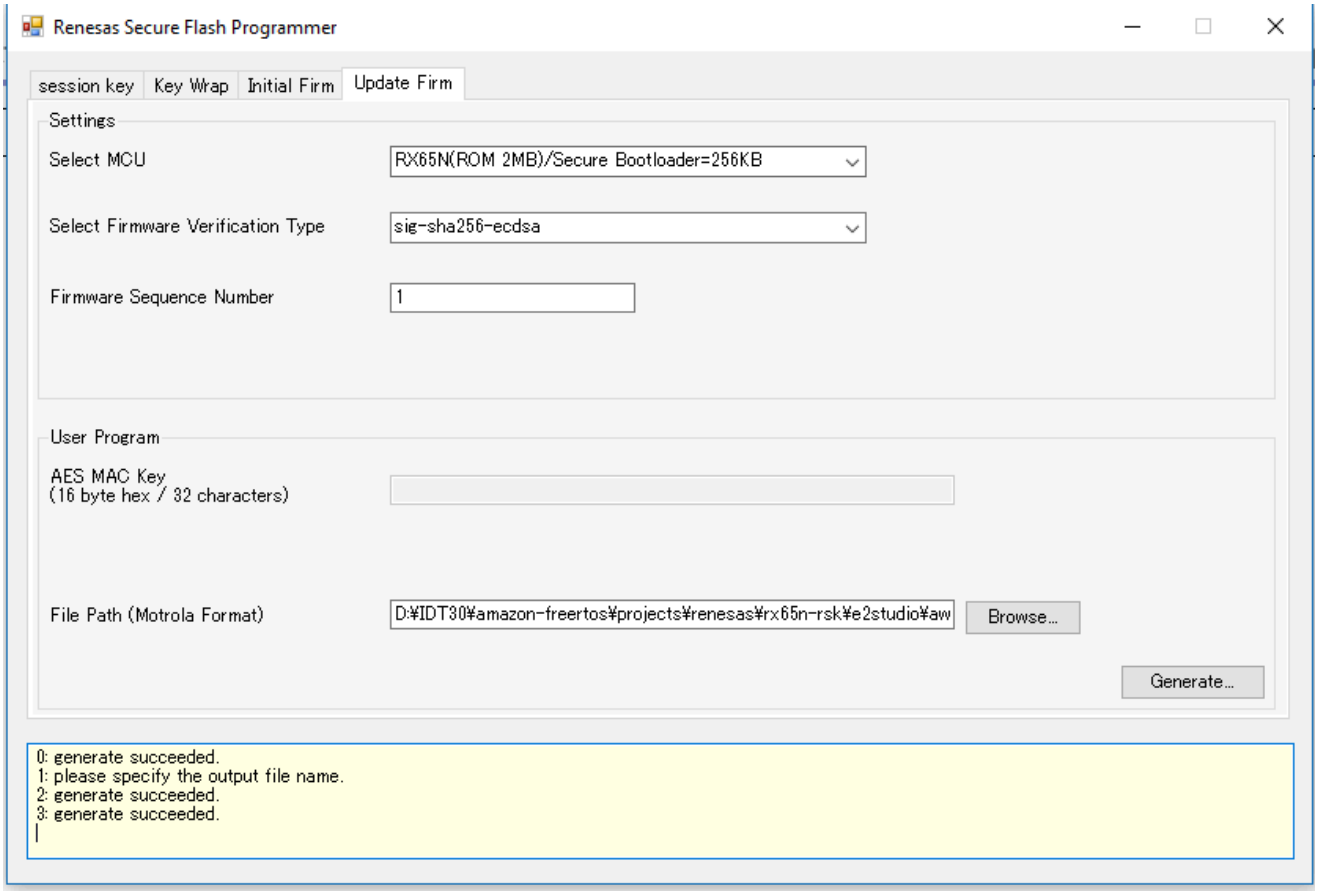
18. Renesas Secure Flash Programmer で `userprog.rsu` ファイルを作成し、ファームウェアのバージョンを更新します。

    a. `Amazon-FreeRTOS-Tools\Renesas Secure Flash Programmer.exe` ファイルを開きます。

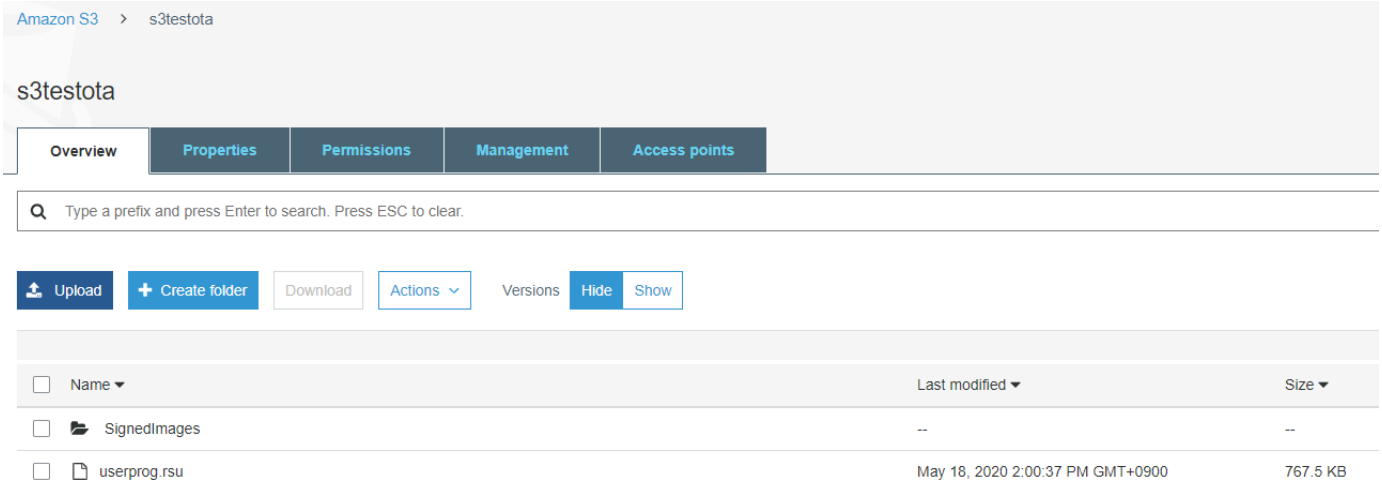
    b. [Update Firm] (ファームの更新) タブを選択し、以下のパラメータを設定します。

    - [File Path] (ファイルパス): `aws_demos.mot` ファイルの場所 (`projects\renesas\rx65n-rsk\e2studio\aws_demos\HardwareDebug`)。

    c. `update _firmware` という名前のディレクトリを作成します。`userprog.rsu` を生成して、`update_firmware` ディレクトリに保存します。生成が成功したことを確認します。



19. 更新を保存する Amazon S3 バケットを作成する のとおり、ファームウェア更新 `userproj.rsu` を Amazon S3 バケットに更新します。



20. RX65N-RSK でファームウェアを更新するジョブを作成します。

    AWS IoT ジョブは、1 つ以上の接続されたデバイスに保留中のジョブを通知するサービスです。ジョブは、多数のデバイスの管理、デバイス上のファームウェアおよびセキュリティ証明書の更新、デバイスの再起動や診断の実行などの管理タスクの実行に使用できます。

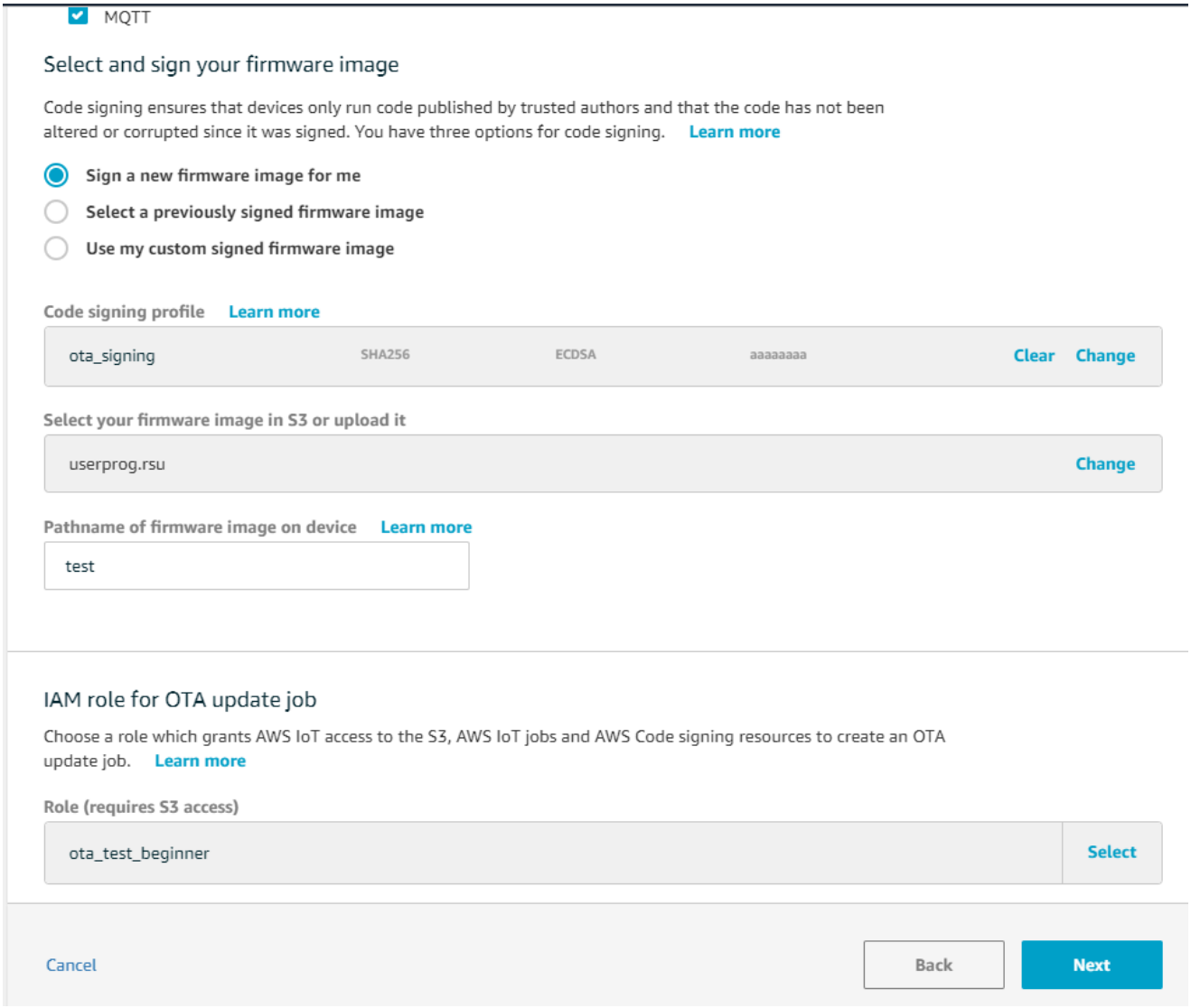
    a. AWS IoT コンソール にサインインします。ナビゲーションペインで、[Manage] (管理)、[Jobs] (ジョブ) の順に選択します。

    b. [Create a job] (ジョブの作成) を選択して、[Create OTA Update job] (OTA 更新ジョブの作成) を選択します。モノを選択して、[Next] (次へ) を選択します。

    c. FreeRTOS OTA 更新ジョブを次のように作成します。

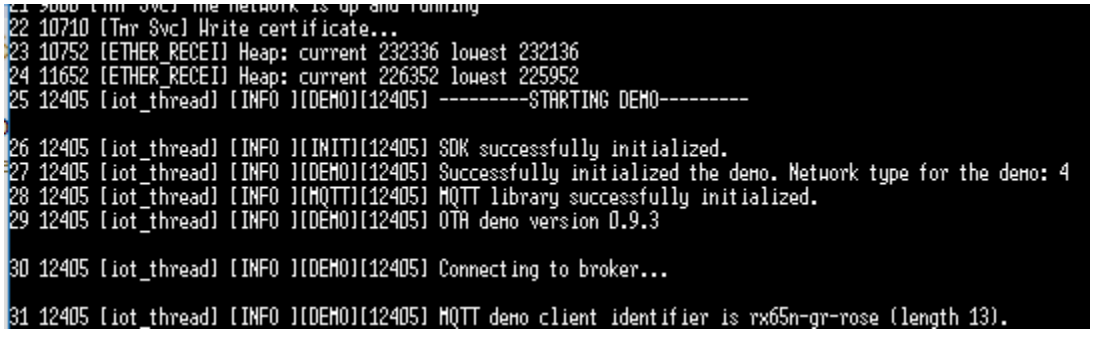
    - [MQTT] を選択します。
    - 前のセクションで作成したコード署名プロファイルを選択します。
    - Amazon S3 バケットにアップロードしたファームウェアイメージを選択します。
    - [Pathname of firmware image on device] (デバイスのファームウェアイメージのパス名) で、**`test`** と入力します。
    - 前のセクションで作成した IAM ロールを選択します。

    d. [Next (次へ)] を選択します。

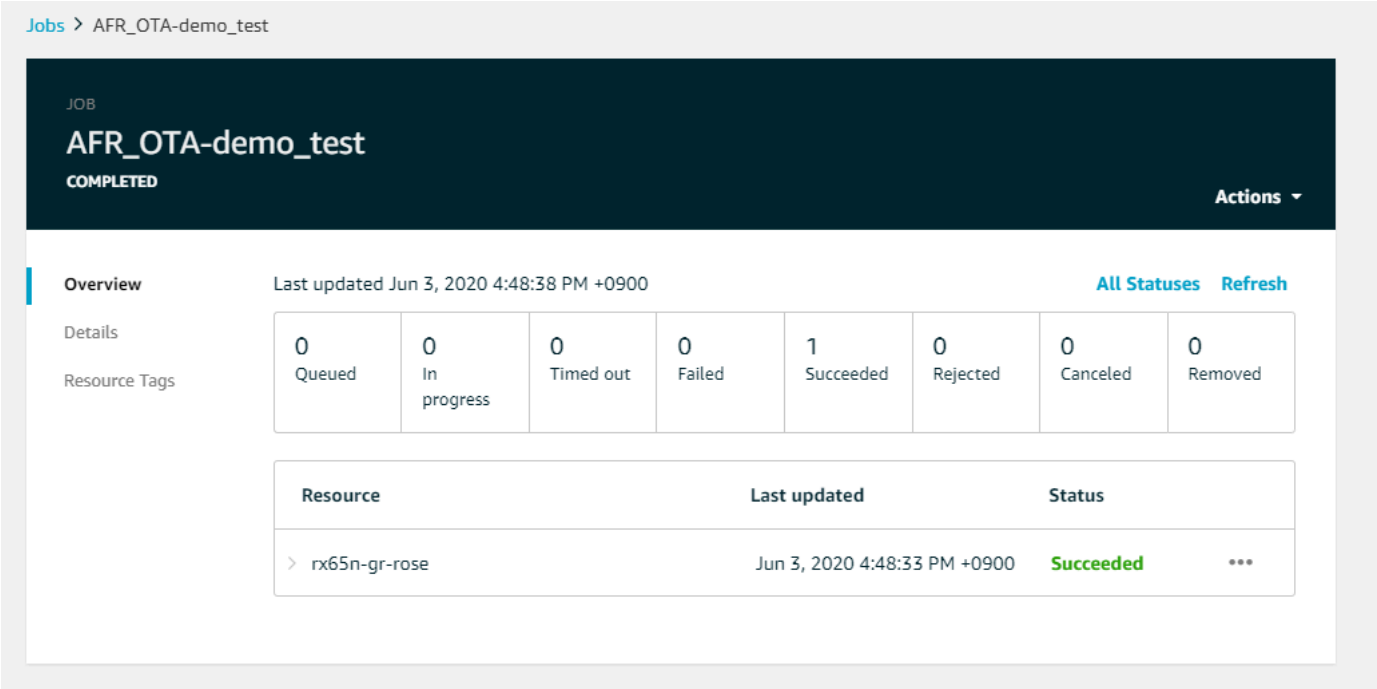


e. ID を入力して、[Create] (作成) を選択します。

21. Tera Term を再度開いて、ファームウェアが OTA デモバージョン 0.9.3 に正常に更新されたことを確認します。



22. AWS IoT コンソールで、ジョブのステータスが成功であることを確認します。



チュートリアル: FreeRTOS Bluetooth Low Energy を使用した Espressif ESP32 での OTA 更新の実行

> **Important**
>
> このリファレンス統合は、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

このチュートリアルでは、Android デバイスの MQTT Bluetooth Low Energy プロキシに接続されている Espressif ESP32 マイクロコントローラーを更新する方法について示します。 AWS IoT Over-the-air (OTA) 更新ジョブを使用してデバイスを更新します。デバイスは、Android デモアプリに入力された Amazon Cognito 認証情報 AWS IoT を使用して に接続します。認可されたオペレータがクラウドから OTA 更新を開始します。デバイスが Android デモアプリを介して接続すると、OTA 更新が開始され、デバイスのファームウェアが更新されます。

FreeRTOS バージョン 2019.06.00 以降メジャーには、Wi-Fi プロビジョニングや AWS IoT サービスへの安全な接続に使用できる Bluetooth Low Energy MQTT プロキシサポートが含まれています。Bluetooth Low Energy 機能を使用すると、Wi-Fi を必要とせずにモバイルデバイスとペアリング

して接続できる低電力デバイスを構築できます。デバイスは、汎用アクセスプロファイル (GAP) と汎用属性 (GATT) プロファイルを使用する Android または iOS の Bluetooth Low Energy SDK を介して接続することにより、MQTT を使用して通信できます。

Bluetooth Low Energy 経由の OTA 更新を許可するステップは次のとおりです。

1. ストレージを設定する: Amazon S3 バケットとポリシーを作成し、更新を実行できるユーザーを設定します。
2. コード署名証明書を作成する: 署名証明書を作成し、ユーザーによるファームウェアの更新の署名を許可します。
3. Amazon Cognito 認証を設定する: 認証情報プロバイダー、ユーザープール、およびユーザープールへのアプリケーションアクセスを作成します。
4. FreeRTOS を設定する: Bluetooth Low Energy、クライアント認証情報、およびコード署名パブリック証明書を設定します。
5. Android アプリを設定する: 認証情報プロバイダー、ユーザープールを設定し、アプリケーションを Android デバイスにデプロイします。
6. OTA 更新スクリプトを実行する: OTA 更新を開始するには、OTA 更新スクリプトを使用します。

更新の仕組みについては、「FreeRTOS 無線通信経由更新」を参照してください。Bluetooth Low Energy MQTT プロキシ機能のセットアップ方法の詳細については、次の記事を参照してください: Espressif ESP32 での FreeRTOS Bluetooth Low Energy の使用 (執筆者 Richard Kang)。

前提条件

このチュートリアルのステップを実行するには、以下のリソースを持っている必要があります。

- ESP32 開発ボード。
- MicroUSB to USB A ケーブル 1 本。
- AWS アカウント (無料利用枠で十分です) 。
- Android v 6.0 以降と Bluetooth バージョン 4.2 以降を搭載した Android 携帯電話。

開発用コンピュータには、次のものが必要です。

- Xtensa ツールチェーンと FreeRTOS のソースコードと例のための十分なディスク容量 (約 500 Mb)。
- インストール済みの Android Studio。

- インストール済みの AWS CLI。
- インストール済みの Python3。
- Python 用 boto3 AWS Software Developer Kit (SDK)。

このチュートリアルのステップは、Xtensa ツールチェーン、ESP-IDF、および FreeRTOS コードがホームディレクトリ内の `/esp` ディレクトリにインストールされていることを前提としています。`~/esp/xtensa-esp32-elf/bin` を `$PATH` 変数に追加する必要があります。

ステップ 1: ストレージを設定する

1. ファームウェアイメージを保持するためにバージョニングを有効にしている 更新を保存する Amazon S3 バケットを作成する。
2. OTA 更新サービスロールを作成する、および次の管理ポリシーをロールに追加します。
   - AWSIotLogging
   - AWSIoTRuleActions
   - AWSIoTThingsRegistration
   - AWSFreeRTOSOTAUpdate
3. OTA 更新を実行できるユーザーを作成します。このユーザーは、アカウント内の IoT デバイスに対するファームウェアアップデートに署名して、それをデプロイできるほか、すべてのデバイスで OTA 更新を実行するためのアクセス権を持っています。アクセスは信頼できるエンティティに限定する必要があります。
4. 「OTA ユーザーポリシーの作成」のステップを実行し、そのポリシーをユーザーにアタッチします。

ステップ 2: コード署名証明書を作成する

1. ファームウェアイメージを保持するために、バージョニングを有効にした Amazon S3 バケットを作成します。
2. ファームウェアの署名に使用できるコード署名証明書を作成します。証明書をインポートするときに、証明書の Amazon リソースネーム (ARN) を書き留めます。

   ```
   aws acm import-certificate --profile=ota-update-user --certificate file://
   ecdsasigner.crt --private-key file://ecdsasigner.key
   ```

   出力例:

```
{
"CertificateArn": "arn:aws:acm:us-east-1:<account>:certificate/<certid>"
}
```

ARN は、後で署名プロファイルを作成するために使用します。必要に応じて、次のコマンドを使用してプロファイルを作成できます。

```
aws signer put-signing-profile --profile=ota-update-user --profile-
name esp32Profile --signing-material certificateArn=arn:aws:acm:us-
east-1:account:certificate/certid --platform AmazonFreeRTOS-Default --signing-
parameters certname=/cert.pem
```

出力例:

```
{
"arn": "arn:aws:signer::<account>:/signing-profiles/esp32Profile"
}
```

ステップ 3: Amazon Cognito 認証の設定

AWS IoT ポリシーを作成する

1. AWS IoT コンソール にサインインします。
2. コンソールの右上隅で、[My Account] (マイアカウント) を選択します。[Account Settings] (アカウント設定) で、12 桁のアカウント ID を書き留めます。
3. 左のナビゲーションペインの [設定] を選択します。[Device data endpoint] (デバイスデータエンドポイント) で、エンドポイントの値を書き留めます。エンドポイントは、xxxxxxxxxxxxxx.iot.us-west-2.amazonaws.com のようになります。この例では、AWS リージョンは「us-west-2」です。
4. 左のナビゲーションペインで [Secure] (安全性) を選択し、[Policies] (ポリシー) を選択してから [Create] (作成) を選択します。アカウントにポリシーがない場合は、「You don't have any policies yet (まだポリシーがありません)」というメッセージが表示され、[Create a policy] (ポリシーの作成) を選択できます。
5. ポリシーの名前を入力します。例えば、「esp32_mqtt_proxy_iot_policy」などです。
6. [Add statements] (ステートメントを追加) セクションで、[Advanced mode] (アドバンストモード) を選択します。次の JSON をポリシーエディタウィンドウにコピーして貼り付けま

す。aws-account-id を自分のアカウント ID に、aws-region を自分のリージョン ([us-west-2] など) に置き換えます。

```
{
  "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:*"
    }
  ]
}
```

7. [Create] (作成) を選択します。

AWS IoT モノを作成する

1. AWS IoT コンソール にサインインします。
2. 左側のナビゲーションペインで、[管理]、[モノ] の順に選択します。
3. 右上隅の [Create] (作成) を選択します。アカウントにモノが登録されていない場合は、「You don't have any things yet (まだモノがありません)」というメッセージが表示され、[Register a thing] (モノの登録) を選択できます。
4. AWS IoT モノの作成ページで、モノを 1 つ作成 を選択します。

5. [Add your device to the thing registry] (Thing Registry にデバイスを追加) ページで、モノの名前 ([esp32-ble] など) を入力します。文字には、英数字、ハイフン (-)、とアンダースコア (_) のみを使用できます。[Next (次へ)] を選択します。
6. [Add a certificate for your thing] (モノに証明書を追加) ページの [Skip certificate and create thing] (証明書をスキップしてモノを作成) から [Create thing without certificate] (証明書なしでモノを作成) を選択します。認証と認可に Amazon Cognito 認証情報を使用する BLE プロキシモバイルアプリケーションを使用しているため、デバイス証明書は必要ありません。

Amazon Cognito アプリクライアントを作成する

1. Amazon Cognito コンソールにサインインします。
2. 右上のナビゲーションバナーで、[Create a user pool] (ユーザープールを作成する) を選択します。
3. プール名を入力します (例えば「esp32_mqtt_proxy_user_pool」)。
4. [Review defaults] を選択します。
5. [App Clients] (アプリクライアント) で、[Add app client] (アプリクライアントの追加) を選択してから、[Add an app client] (アプリクライアントの追加) を選択します。
6. アプリクライアント名を入力します (例えば「mqtt_app_client」)。
7. [Generate client secret] (クライアントシークレットを生成) が選択されていることを確認します。
8. [Create app client] を選択します。
9. [プールの詳細に戻る] を選択します。
10. ユーザープールの [Review] (確認) ページで、[Create pool] (プールの作成) を選択します。[Your user pool was created successfully] (ユーザープールは正常に作成されました) というメッセージが表示されます。プール ID を書き留めます。
11. ナビゲーションペインで、[App clients] (アプリクライアント) を選択します。
12. [Show Details] (詳細を表示) を選択します。アプリクライアント ID およびアプリクライアントシークレットを書き留めます。

Amazon Cognito アイデンティティプールを作成する

1. Amazon Cognito コンソールにサインインします。
2. [Create new identity pool] を選択します。

3. アイデンティティプールの名前を入力します (例えば「mqtt_proxy_identity_pool」)。
4. [Authentication providers] (認証プロバイダー) を展開します。
5. [Cognito] タブを選択します。
6. 前のステップで書き留めておいたユーザープール ID とアプリクライアント ID を入力します。
7. [プールの作成] を選択します。
8. 次のページで、認証されたアイデンティティおよび 認証されていないアイデンティティの新しいロールを作成するために、[Allow] (許可) を選択します。
9. アイデンティティプールの ID を書き留めます。形式は `us-east-1:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` です。

認証されたアイデンティティに IAM ポリシーをアタッチします。

1. Amazon Cognito コンソールを開きます。
2. 今しがた作成したアイデンティティプールを選択します (例えば「mqtt_proxy_identity_pool」)。
3. [Edit identity pool] (ID プールの編集) をクリックします。
4. 認証されたロールに割り当てられた IAM ロールを書き留めます (例えば「Cognito_mqtt_proxy_identity_poolAuth_Role」)。
5. IAM コンソールを開きます。
6. ナビゲーションペインで [Roles (ロール)] を選択します。
7. 割り当てられたロール (例えば「Cognito_mqtt_proxy_identity_poolAuth_Role」) を検索して選択します。
8. [Add inline policy] (インラインポリシーの追加) を選択し、次に [JSON] を選択します。
9. 以下のポリシーを入力します。

```
{
       "Version": "2012-10-17",
       "Statement": [
       {
          "Effect": "Allow",
          "Action": [
             "iot:AttachPolicy",
             "iot:AttachPrincipalPolicy",
             "iot:Connect",
             "iot:Publish",
             "iot:Subscribe"
```

```
            ],
            "Resource": "*"
        }]
    }
```

10. [ポリシーの確認] を選択します。
11. ポリシー名を入力します (例えば「mqttProxyCognitoPolicy」)。
12. [Create policy] を選択します。

ステップ 4: Amazon FreeRTOS を設定する

1. FreeRTOS GitHub リポジトリから、最新バージョンの Amazon FreeRTOS コードをダウンロードします。
2. OTA アップデートのデモを有効にするには、Espressif ESP32-DevKitC と ESP-WROVER-KIT の開始方法 のステップに従います。
3. 次のファイルで、これらの追加変更を行います。

   a. `vendors/espressif/boards/esp32/aws_demos/config_files/aws_demo_config.h` を開き、`CONFIG_OTA_UPDATE_DEMO_ENABLED` を定義します。

   b. `vendors/espressif/boards/esp32/aws_demos/common/config_files/aws_demo_config.h` を開き、`democonfigNETWORK_TYPES` を `AWSIOT_NETWORK_TYPE_BLE` に変更します。

   c. `demos/include/aws_clientcredential.h` を開き、`clientcredentialMQTT_BROKER_ENDPOINT` のエンドポイント URL を入力します。

      `clientcredentialIOT_THING_NAME` のモノの名前を入力します (例えば「esp32-ble」)。Amazon Cognito 認証情報を使用する場合、証明書を追加する必要はありません。

   d. `vendors/espressif/boards/esp32/aws_demos/config_files/aws_iot_network_config.h` を開き、`configSUPPORTED_NETWORKS` と `configENABLED_NETWORKS` を変更して `AWSIOT_NETWORK_TYPE_BLE` のみを含めるようにします。

   e. `vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/ota_demo_config.h` ファイルを開き、証明書を入力します。

      ```
      #define otapalconfigCODE_SIGNING_CERTIFICATE [] = "your-certificate-key";
      ```

アプリケーションが起動し、次のデモ版を出力します。

```
11 13498 [iot_thread] [INFO ][DEMO][134980] Successfully initialized the demo.
 Network type for the demo: 2
12 13498 [iot_thread] [INFO ][MQTT][134980] MQTT library successfully initialized.
13 13498 [iot_thread] OTA demo version 0.9.20
14 13498 [iot_thread] Creating MQTT Client...
```

ステップ 5: Android アプリケーションを設定する

1. Android Bluetooth Low Energy SDK とサンプルアプリケーションを amazon-freertos-ble-android-sdk GitHub レポからダウンロードします。
2. ファイル `app/src/main/res/raw/awsconfiguration.json` を開き、次の JSON サンプルの手順を使用して、Pool Id (プール ID)、Region (リージョン)、AppClientID (アプリクライアント ID)、および AppClientSecret (アプリクライアントシークレット) を入力します。

```
{
  "UserAgent": "MobileHub/1.0",
  "Version": "1.0",
  "CredentialsProvider": {
    "CognitoIdentity": {
      "Default": {
        "PoolId": "Cognito->Manage Identity Pools->Federated Identities-
>mqtt_proxy_identity_pool->Edit Identity Pool->Identity Pool ID",
        "Region": "Your region (for example us-east-1)"
      }
    }
  },

  "IdentityManager": {
    "Default": {}
  },

  "CognitoUserPool": {
    "Default": {
      "PoolId": "Cognito-> Manage User Pools -> esp32_mqtt_proxy_user_pool ->
 General Settings -> PoolId",
      "AppClientId": "Cognito-> Manage User Pools -> esp32_mqtt_proxy_user_pool ->
 General Settings -> App clients ->Show Details",
```

```
      "AppClientSecret": "Cognito-> Manage User Pools -> esp32_mqtt_proxy_user_pool
 -> General Settings -> App clients ->Show Details",
      "Region": "Your region (for example us-east-1)"
    }
  }
}
```

3. app/src/main/java/software/amazon/freertos/DemoConstants.java を開き、前もって作成したポリシー名 (例: *esp32_mqtt_proxy_iot_policy*) とリージョン (例: *us-east-1*)) を入力します。
4. デモアプリケーションを構築してインストールします。
   a. Android Studio で、[Build] (構築) を選択してから [Make Module app] (モジュールアプリの作成) を選択します。
   b. [Run] (実行) を選択して [Run app] (アプリケーションを実行) を選択します。Android Studio の logcat ウィンドウペインに移動して、ログメッセージをモニタリングできます。
   c. Android デバイスで、ログイン画面からアカウントを作成します。
   d. ユーザーを作成します。ユーザーが既に存在する場合は、認証情報を入力します。
   e. Amazon FreeRTOS デモによるデバイスの位置情報へのアクセスを許可します。
   f. Bluetooth Low Energy デバイスをスキャンします。
   g. 見つかったデバイスのスライダーを [On] (オン) にします。
   h. ESP32 のシリアルポートデバッグコンソールで y を押します。
   i. [Pair & Connect] (ペアリング & 接続) を選択します。
5. 接続が確立されると、[More...] (詳細...) リンクがアクティブになります。接続が完了すると、次のように Android デバイスの logcat で接続状態が「BLE_CONNECTED」に変更されます。

```
2019-06-06 20:11:32.160 23484-23497/software.amazon.freertos.demo I/FRD: BLE
 connection state changed: 0; new state: BLE_CONNECTED
```

6. メッセージを送信する前に、Amazon FreeRTOS デバイスと Android デバイスが MTU をネゴシエートします。logcat に次のような出力が表示されます。

```
2019-06-06 20:11:46.720 23484-23497/software.amazon.freertos.demo I/FRD:
 onMTUChanged : 512 status: Success
```

7. デバイスがアプリケーションに接続し、MQTT プロキシを使用して MQTT メッセージの送信を開始します。デバイスが通信できることを確認するには、MQTT_CONTROL の特性データ値が次のように 01 に変更されていることを確認します。

```
2019-06-06 20:12:28.752 23484-23496/software.amazon.freertos.demo D/FRD: <-<-<-
 Writing to characteristic: MQTT_CONTROL with data: 01
2019-06-06 20:12:28.839 23484-23496/software.amazon.freertos.demo D/FRD:
 onCharacteristicWrite for: MQTT_CONTROL; status: Success; value: 01
```

8. デバイスがペアリングされると、ESP32 コンソールにプロンプトが表示されます。BLE を有効にするには、y を押します。このステップを実行するまで、デモは機能しません。

```
E (135538) BT_GATT: GATT_INSUF_AUTHENTICATION: MITM Required
W (135638) BT_L2CAP: l2cble_start_conn_update, the last connection update command
 still pending.
E (135908) BT_SMP: Value for numeric comparison = 391840
15 13588 [InputTask] Numeric comparison:391840
16 13589 [InputTask] Press 'y' to confirm
17 14078 [InputTask] Key accepted
W (146348) BT_SMP: FOR LE SC LTK IS USED INSTEAD OF STK
18 16298 [iot_thread] Connecting to broker...
19 16298 [iot_thread] [INFO ][MQTT][162980] Establishing new MQTT connection.
20 16298 [iot_thread] [INFO ][MQTT][162980] (MQTT connection 0x3ffd5754, CONNECT
 operation 0x3ffd586c) Waiting for operation completion.
21 16446 [iot_thread] [INFO ][MQTT][164450] (MQTT connection 0x3ffd5754, CONNECT
 operation 0x3ffd586c) Wait complete with result SUCCESS.
22 16446 [iot_thread] [INFO ][MQTT][164460] New MQTT connection 0x3ffc0ccc
 established.
23 16446 [iot_thread] Connected to broker.
```

ステップ 6: OTA 更新スクリプトを実行する

1. 前提条件をインストールするには、次のコマンドを実行します。

```
pip3 install boto3
```

```
pip3 install pathlib
```

2. FreeRTOS アプリケーションバージョンを demos/include/aws_application_version.h にインクリメントします。

3. 新しい .bin ファイルを構築します。
4. Python スクリプト start_ota.py をダウンロードします。スクリプトのヘルプの内容を表示するには、ターミナルウィンドウで次のコマンドを実行します。

```
python3 start_ota.py -h
```

次のようなものが表示されます。

```
usage: start_ota.py [-h] --profile PROFILE [--region REGION]
                    [--account ACCOUNT] [--devicetype DEVICETYPE] --name NAME
                    --role ROLE --s3bucket S3BUCKET --otasigningprofile
                    OTASIGNINGPROFILE --signingcertificateid
                    SIGNINGCERTIFICATEID [--codelocation CODELOCATION]
Script to start OTA update
optional arguments:
-h, --help            show this help message and exit
--profile PROFILE     Profile name created using aws configure
--region REGION       Region
--account ACCOUNT     Account ID
--devicetype DEVICETYPE thing|group
--name NAME           Name of thing/group
--role ROLE           Role for OTA updates
--s3bucket S3BUCKET   S3 bucket to store firmware updates
--otasigningprofile OTASIGNINGPROFILE
                      Signing profile to be created or used
--signingcertificateid SIGNINGCERTIFICATEID
                      certificate id (not arn) to be used
--codelocation CODELOCATION
                      base folder location (can be relative)
```

5. 提供された AWS CloudFormation テンプレートを使用してリソースを作成した場合は、次のコマンドを実行します。

```
python3 start_ota_stream.py --profile otausercf --name esp32-ble --role
 ota_ble_iot_role-sample --s3bucket afr-ble-ota-update-bucket-sample --
otasigningprofile abcd --signingcertificateid certificateid
```

ESP32 デバッグコンソールで更新がスタートします。

```
38 2462 [OTA Task] [prvParseJobDoc] Job was accepted. Attempting to start transfer.
---
```

```
49 2867 [OTA Task] [prvIngestDataBlock] Received file block 1, size 1024
50 2867 [OTA Task] [prvIngestDataBlock] Remaining: 1290
51 2894 [OTA Task] [prvIngestDataBlock] Received file block 2, size 1024
52 2894 [OTA Task] [prvIngestDataBlock] Remaining: 1289
53 2921 [OTA Task] [prvIngestDataBlock] Received file block 3, size 1024
54 2921 [OTA Task] [prvIngestDataBlock] Remaining: 1288
55 2952 [OTA Task] [prvIngestDataBlock] Received file block 4, size 1024
56 2953 [OTA Task] [prvIngestDataBlock] Remaining: 1287
57 2959 [iot_thread] State: Active  Received: 5   Queued: 5   Processed: 5
 Dropped: 0
```

6. OTA 更新が完了すると、OTA 更新プロセスの要求に従ってデバイスが再起動します。次に、更新されたファームウェアを使用した接続が試行されます。アップグレードが成功すると、次のように更新されたファームウェアがアクティブとしてマークされ、コンソールに更新されたバージョンが表示されます。

```
13 13498 [iot_thread] OTA demo version 0.9.21
```

AWS IoT Device Shadow デモアプリケーション

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

序章

このデモでは、 AWS IoT Device Shadow ライブラリを使用して AWS Device Shadow サービスに接続する方法を示します。を使用して MQTT ブローカーへの TLS (相互認証) による AWS IoT MQTT 接続coreMQTT ライブラリを確立し、coreJSON AWS ライブラリパーサーを使用して Shadow サービスから受信したシャドウドキュメントを解析します。デモでは、シャドウドキュメントの更新方法やシャドウドキュメントの削除方法など、基本的なシャドウオペレーションについて説明します。また、 AWS IoT Device Shadow サービスから送信されるメッセージ (シャドウ `/update` や `/update/delta` メッセージなど) を処理するために coreMQTT ライブラリを使ってコールバック関数を登録する方法も示します。

このデモは、シャドウドキュメント (状態) の更新要求と更新レスポンスが同じアプリケーションによって行われるため、学習演習としてのみ意図されています。現実的な本番シナリオでは、デバイスが現在接続されていない場合でも、外部アプリケーションによってデバイスの状態の更新がリモートで要求されます。デバイスは、接続時に更新要求を承認します。

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。

機能

デモでは、シャドウ /update と /update/delta コールバックを示す一連の例をループする 1 つのアプリケーションタスクを作成して、リモートデバイスの状態を切り替えます。新しい desired 状態のシャドウアップデートを送信し、デバイスが新しい desired 状態に応答して reported を変更するまで待ちます。さらに、シャドウ /update コールバックは、変化するシャドウの状態を出力するために使用されます。このデモでは、MQTT ブローカーへの安全な AWS IoT MQTT 接続も使用し、デバイスシャドウに powerOn状態があることを前提としています。

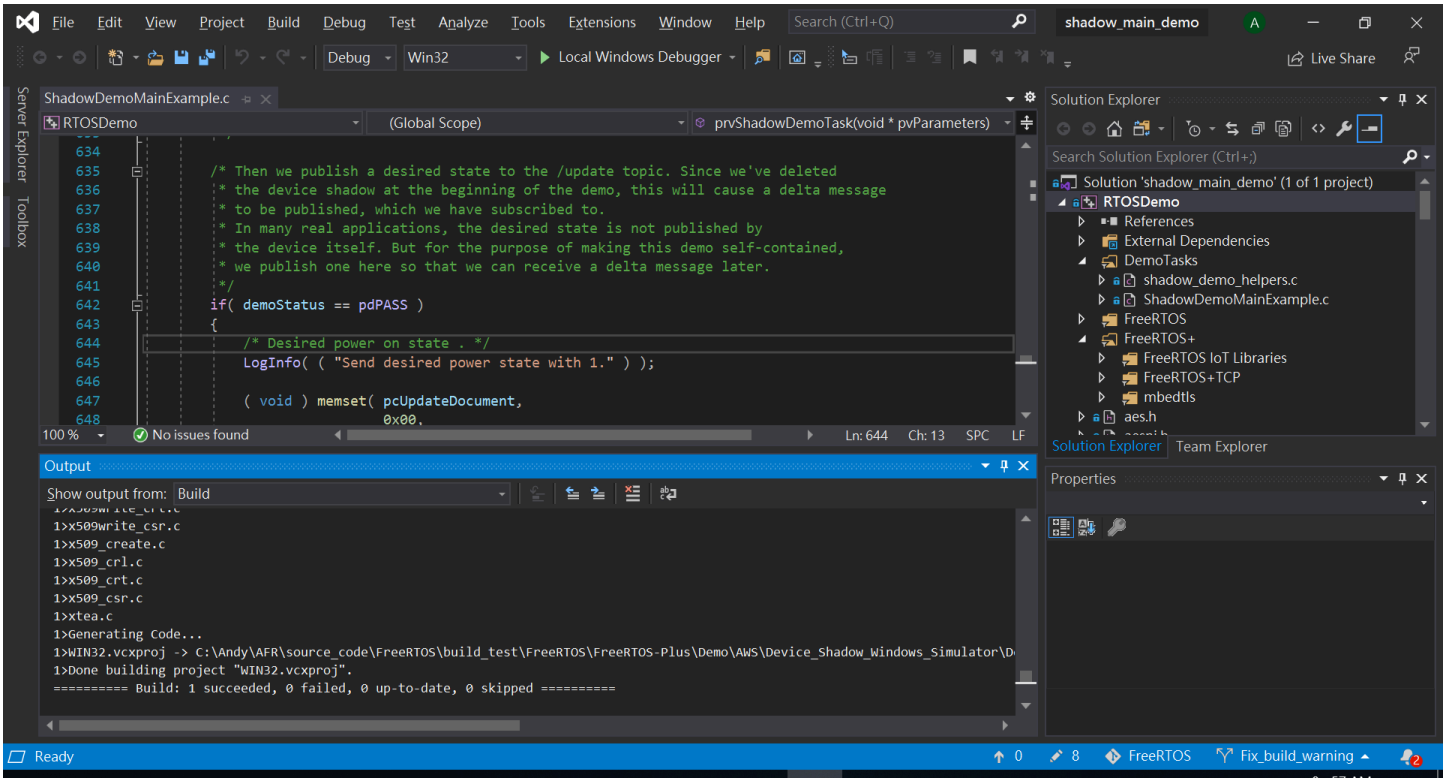
このデモでは以下のオペレーションを実行します。

1. shadow_demo_helpers.c のヘルパー関数を使用して MQTT 接続を確立します。
2. AWS IoT Device Shadow ライブラリで定義されたマクロを使用して、デバイスシャドウオペレーションの MQTT トピック文字列を組み立てます。
3. デバイスシャドウの削除用に使用する MQTT トピックに公開して、既存のデバイスシャドウを削除します。
4. shadow_demo_helpers.c のヘルパー関数を使って、/update/delta、/update/accepted、/update/rejected の MQTT トピックをサブスクライブします。
5. shadow_demo_helpers.c のヘルパー関数を使って powerOn の望ましい状態を公開します。これにより、デバイスに /update/delta メッセージが送信されます。
6. で受信 MQTT メッセージを処理しprvEventCallback、Device Shadow ライブラリ () で定義された関数を使用して、メッセージが AWS IoT デバイスシャドウに関連しているかどうかを判断しますShadow_MatchTopic。メッセージがデバイスシャドウ /update/delta メッセージの場合、メインのデモ関数が 2 つ目のメッセージを発行して、報告された状態を powerOn に更新します。/update/accepted メッセージを受信した場合、以前更新メッセージで発行されたものと clientToken が同じか確認します。これでデモは終了です。

```
82 9136 [ShadowDemo] [INFO] [SHADOW] [prvShadowDemoTask:641] 83 9136 [ShadowDemo] Send desired power state with 1.84 9136 [ShadowDemo]
85 9296 [ShadowDemo] [INFO] [SHADOW] [prvEventCallback:482] 86 9296 [ShadowDemo] pPublishInfo->pTopicName:$aws/things/testClient16:34:41/shadow/update/delta.87 9296 [Shad
owDemo]
88 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateDeltaHandler:256] 89 9296 [ShadowDemo] /update/delta json payload:{"version":1,"timestamp":1602751002,"state":{"powerOn":1}
,"metadata":{"powerOn":{"timestamp":1602751002}},"clientToken":"009136"}.90 9296 [ShadowDemo]
91 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateDeltaHandler:298] 92 9296 [ShadowDemo] version: 193 9296 [ShadowDemo]
94 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateDeltaHandler:308] 95 9296 [ShadowDemo] version:1, ulCurrentVersion:0
96 9296 [ShadowDemo]
97 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateDeltaHandler:342] 98 9296 [ShadowDemo] The new power on state newState:1, ulCurrentPowerOnState:0
99 9296 [ShadowDemo]
100 9296 [ShadowDemo] [INFO] [SHADOW] [prvEventCallback:482] 101 9296 [ShadowDemo] pPublishInfo->pTopicName:$aws/things/testClient16:34:41/shadow/update/accepted.102 9296
 [ShadowDemo]
103 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:376] 104 9296 [ShadowDemo] /update/accepted json payload:{"state":{"desired":{"powerOn":1}},"metadata":{"d
esired":{"powerOn":{"timestamp":1602751002}}},"version":1,"timestamp":1602751002,"clientToken":"009136"}.105 9296 [ShadowDemo]
106 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:424] 107 9296 [ShadowDemo] clientToken: 009136108 9296 [ShadowDemo]
109 9296 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:429] 110 9296 [ShadowDemo] receivedToken:9136, clientToken:0
111 9296 [ShadowDemo]
112 9296 [ShadowDemo] [WARN] [SHADOW] [prvUpdateAcceptedHandler:442] 113 9296 [ShadowDemo] The received clientToken=9136 is not identical with the one=0 we sent 114 9296
[ShadowDemo]
115 9696 [ShadowDemo] [INFO] [SHADOW] [prvShadowDemoTask:670] 116 9696 [ShadowDemo] Report to the state change: 1117 9696 [ShadowDemo]
118 9856 [ShadowDemo] [INFO] [SHADOW] [prvEventCallback:482] 119 9856 [ShadowDemo] pPublishInfo->pTopicName:$aws/things/testClient16:34:41/shadow/update/accepted.120 9856
 [ShadowDemo]
121 9856 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:376] 122 9856 [ShadowDemo] /update/accepted json payload:{"state":{"reported":{"powerOn":1}},"metadata":{"
reported":{"powerOn":{"timestamp":1602751003}}},"version":2,"timestamp":1602751003,"clientToken":"009696"}.123 9856 [ShadowDemo]
124 9856 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:424] 125 9856 [ShadowDemo] clientToken: 009696126 9856 [ShadowDemo]
127 9856 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:429] 128 9856 [ShadowDemo] receivedToken:9696, clientToken:9696
129 9856 [ShadowDemo]
130 9856 [ShadowDemo] [INFO] [SHADOW] [prvUpdateAcceptedHandler:437] 131 9856 [ShadowDemo] Received response from the device shadow. Previously published update with clie
ntToken=9696 has been accepted. 132 9856 [ShadowDemo]
133 10256 [ShadowDemo] [INFO] [SHADOW] [prvShadowDemoTask:698] 134 10256 [ShadowDemo] Start to unsubscribe shadow topics and disconnect from MQTT.
135 10256 [ShadowDemo]
136 12036 [ShadowDemo] [INFO] [SHADOW] [prvShadowDemoTask:747] 137 12036 [ShadowDemo] Demo completed successfully.138 12036 [ShadowDemo]
139 12036 [ShadowDemo] [INFO] [SHADOW] [prvShadowDemoTask:756] 140 12036 [ShadowDemo] Deleting Shadow Demo task.141 12036 [ShadowDemo]
```

デモはファイル *freertos*/demos/device_shadow_for_aws/shadow_demo_main.c または GitHub で手に入ります。

次のスクリーンショットは、デモが成功したときに予想される出力です。



AWS IoT MQTT ブローカーに接続する

AWS IoT MQTT ブローカーに接続するには、 `MQTT_Connect()`と同じ方法を使用しますcoreMQTT Mutual Authentication デモ。

シャドウキュメントを削除する

シャドウドキュメントを削除するには、 AWS IoT Device Shadow ライブラリで定義されたマクロを使用して、空のメッセージxPublishToTopicで を呼び出します。これは `MQTT_Publish` を使用して `/delete` トピックに公開します。次のコードセクションでは、これを `prvShadowDemoTask` 関数で実行する方法を示します。

```
/* First of all, try to delete any Shadow document in the cloud. */
returnStatus = PublishToTopic( SHADOW_TOPIC_STRING_DELETE( THING_NAME ),
                               SHADOW_TOPIC_LENGTH_DELETE( THING_NAME_LENGTH ),
                               pcUpdateDocument,
                               0U );
```

シャドウトピックをサブスクライブする

Device Shadow トピックをサブスクライブして、シャドウの変更に関する通知を AWS IoT ブローカーから受信します。Device Shadow のトピックは、Device Shadow ライブラリで定義されているマクロによって組み立てられます。次のコードセクションでは、これを `prvShadowDemoTask` 関数で実行する方法を示します。

```
/* Then try to subscribe shadow topics. */
if( returnStatus == EXIT_SUCCESS )
{
    returnStatus = SubscribeToTopic(
                     SHADOW_TOPIC_STRING_UPDATE_DELTA( THING_NAME ),
                     SHADOW_TOPIC_LENGTH_UPDATE_DELTA( THING_NAME_LENGTH ) );
}

if( returnStatus == EXIT_SUCCESS )
{
    returnStatus = SubscribeToTopic(
                     SHADOW_TOPIC_STRING_UPDATE_ACCEPTED( THING_NAME ),
                     SHADOW_TOPIC_LENGTH_UPDATE_ACCEPTED( THING_NAME_LENGTH ) );
}
```

```
if( returnStatus == EXIT_SUCCESS )
{
    returnStatus = SubscribeToTopic(
                      SHADOW_TOPIC_STRING_UPDATE_REJECTED( THING_NAME ),
                      SHADOW_TOPIC_LENGTH_UPDATE_REJECTED( THING_NAME_LENGTH ) );
}
```

シャドウの更新を送信する

シャドウアップデートを送信するために、Device Shadow ライブラリで定義されるマクロを使用して、JSON 形式のメッセージ付きの xPublishToTopic を呼び出します。これは MQTT_Publish を使用して /delete トピックに公開します。次のコードセクションでは、これを prvShadowDemoTask 関数で実行する方法を示します。

```
#define SHADOW_REPORTED_JSON    \
    "{"                         \
    "\"state\":{"               \
    "\"reported\":{"            \
    "\"powerOn\":%01d"          \
    "}"                         \
    "},"                        \
    "\"clientToken\":\"%06lu\"" \
    "}"
snprintf( pcUpdateDocument,
          SHADOW_REPORTED_JSON_LENGTH + 1,
          SHADOW_REPORTED_JSON,
           ( int ) ulCurrentPowerOnState,
           ( long unsigned ) ulClientToken );

xPublishToTopic( SHADOW_TOPIC_STRING_UPDATE( THING_NAME ),
                 SHADOW_TOPIC_LENGTH_UPDATE( THING_NAME_LENGTH ),
                 pcUpdateDocument,
                 ( SHADOW_DESIRED_JSON_LENGTH + 1 ) );
```

シャドウデルタメッセージとシャドウ更新メッセージを処理する

ユーザーコールバック関数は MQTT_Init 関数を使って coreMQTT クライアントライブラリに登録され、受信パケットイベントを通知します。コールバック関数については、GitHub の「prvEventCallback」を参照してください。

コールバック関数は受信パケットが MQTT_PACKET_TYPE_PUBLISH のタイプであることを確認し、Device Shadow ライブラリ API Shadow_MatchTopic を使用して、受信メッセージがシャドウメッセージであることを確認します。

受信メッセージが ShadowMessageTypeUpdateDelta タイプのシャドウメッセージである場合、prvUpdateDeltaHandler を呼び出してこのメッセージを処理します。ハンドラ prvUpdateDeltaHandler が coreJSON ライブラリを使用してメッセージを解析し、powerOn 状態のデルタ値を取得し、これをローカルで保持されているデバイスの現在の状態と比較します。これらが異なる場合、シャドウドキュメントの powerOn 状態の新しい値を反映するようにローカルデバイスの状態が更新されます。

受信メッセージが ShadowMessageTypeUpdateAccepted タイプのシャドウメッセージである場合、prvUpdateAcceptedHandler を呼び出してこのメッセージを処理します。ハンドラ prvUpdateAcceptedHandler は coreJSON ライブラリを使用してメッセージを解析し、メッセージから clientToken を取得します。このハンドラ関数は、JSON メッセージのクライアントトークンが、アプリケーションで使用されるクライアントトークンと一致するかチェックします。一致しない場合、関数は警告メッセージをログに記録します。

セキュアソケットエコークライアントのデモ

> **Important**
>
> このデモは、非推奨の Amazon-FreeRTOS リポジトリでホストされています。新しいプロジェクトを作成するときは、ここから始めることをお勧めします。現在非推奨の Amazon-FreeRTOS リポジトリをベースにした既存の FreeRTOS プロジェクトが既にある場合は、「Amazon FreeRTOS Github リポジトリ移行ガイド」を参照してください。

次の例では、1 つの RTOS タスクを使用します。この例のソースコードは demos/tcp/aws_tcp_echo_client_single_task.c にあります。

開始する前に、マイクロコントローラーに FreeRTOS をダウンロードしてあること、FreeRTOS デモプロジェクトを構築して実行してあることを確認します。FreeRTOS は GitHub からクローンを作成またはダウンロードできます。手順については、README.md ファイルを参照してください。

デモを実行するには

> **Note**
>
> FreeRTOS デモをセットアップして実行するには、FreeRTOS の使用を開始する の手順に従います。
> TCP サーバーとクライアントのデモは、現在、Cypress CYW943907AEVAL1F および CYW954907AEVAL1F 開発キットではサポートされていません。

1. FreeRTOS 移植ガイドの TLS エコーサーバーのセットアップの指示に従ってください。

   TLS Echo Server がポート 9000 で実行され、リッスンします。

   セットアップ中に 4 つのファイルが生成されます。

   - `client.pem` (クライアント証明書)
   - `client.key` (クライアントのプライベートキー)
   - `server.pem` (サーバー証明書)
   - `server.key` (サーバーのプライベートキー)
2. ツール `tools/certificate_configuration/CertificateConfigurator.html` を使ってクライアント証明書 (`client.pem`) とクライアントのプライベートキー (`client.key`) を `aws_clientcredential_keys.h` にコピーします。
3. `FreeRTOSConfig.h` ファイルを開きます。
4. `configECHO_SERVER_ADDR0`、`configECHO_SERVER_ADDR1`、`configECHO_SERVER_ADDR2`、`configECHO_SERVER_ADDR3` の各変数を、TLS Echo Server の実行場所の IP アドレスを構成する 4 桁の整数に設定します。
5. `configTCP_ECHO_CLIENT_PORT` 変数を、TLS Echo Server がリッスンしているポートの `9000` に設定します。
6. `configTCP_ECHO_TASKS_SINGLE_TASK_TLS_ENABLED` 変数を 1 に設定します。
7. ツール `tools/certificate_configuration/PEMfileToCString.html` を使用して、サーバー証明書 (`server.pem`) を `aws_tcp_echo_client_single_task.c` ファイル内の `cTlsECHO_SERVER_CERTIFICATE_PEM` にコピーします。
8. *`freertos`*`/vendors/`*`vendor`*`/boards/`*`board`*`/aws_demos/config_files/aws_demo_config.h` を開き、`#define CONFIG_CORE_MQTT_MUTUAL_AUTH_DEMO_ENABLED` をコメント

アウトして CONFIG_OTA_MQTT_UPDATE_DEMO_ENABLED または CONFIG_OTA_HTTP_UPDATE_DEMO_ENABLED を定義します。

マイクロコントローラーと TLS Echo Server は、同じネットワーク上に配置します。デモが始まると (main.c)、Received correct string from echo server というログメッセージが表示されます。

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。