
Amazon FreeRTOS

ユーザーガイド



Amazon FreeRTOS: ユーザーガイド

Copyright © 2019 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

Amazon FreeRTOS とは	1
Amazon FreeRTOS アーキテクチャ	1
FreeRTOS カーネル	2
FreeRTOS カーネルの基礎	2
FreeRTOS カーネルスケジューラ	2
メモリ管理	2
タスク間の調整	3
ソフトウェアタイマー	7
低電力サポート	7
Amazon FreeRTOS ライブラリ	7
無線による更新	7
無線による更新の前提条件	8
OTA チュートリアル	23
OTA 更新マネージャサービス	45
アプリケーションへの OTA エージェントの統合	46
OTA セキュリティ	48
OTA のトラブルシューティング	49
Amazon FreeRTOS ソースコードのダウンロード	55
Amazon FreeRTOS コンソール	55
Amazon FreeRTOS コンソール	55
事前定義された Amazon FreeRTOS 構成	55
カスタム Amazon FreeRTOS 設定	56
クイック接続ワークフロー	57
開発ワークフロー	57
その他のリソース	57
Amazon FreeRTOS の使用開始	59
最初のステップ	59
ボード固有の入門ガイド	59
トラブルシューティング	59
Amazon FreeRTOS - 認定済みハードウェアプラットフォーム	60
最初のステップ	60
AWS アカウントとアクセス許可の設定	61
AWS IoT で MCU ボードを登録	61
Amazon FreeRTOS のダウンロード	64
Amazon FreeRTOS デモを設定する	64
トラブルシューティング	66
一般的なトラブルシューティングの開始方法のヒント	66
ターミナルエミュレータをインストールする	66
Cypress CYW943907AEVAL1F 開発キット	67
Amazon FreeRTOS デモプロジェクトを構築および実行する	67
Cypress CYW954907AEVAL1F 開発キット	68
Amazon FreeRTOS デモプロジェクトを構築および実行する	67
Espressif ESP32-DevKitC と ESP-WROVER-KIT	69
前提条件	69
Espressif ハードウェアのセットアップ	70
環境をセットアップする	70
Amazon FreeRTOS をダウンロードして設定する	71
Amazon FreeRTOS デモプロジェクトを構築および実行する	73
トラブルシューティング	74
Infineon XMC4800 IoT 接続キット	79
環境をセットアップする	79
Amazon FreeRTOS デモプロジェクトを構築および実行する	80
トラブルシューティング	82
MediaTek MT7697Hx Development Kit	82

環境をセットアップする	83
Keil MDK を使用して Amazon FreeRTOS デモプロジェクトをビルドおよび実行する	83
トラブルシューティング	84
Microchip Curiosity PIC32MZEF	85
Microchip Curiosity PIC32MZEF ハードウェアのセットアップ	85
環境をセットアップする	86
Amazon FreeRTOS デモプロジェクトを構築および実行する	87
トラブルシューティング	88
Nordic nRF52840-DK	88
Nordic ハードウェアのセットアップ	89
環境をセットアップする	89
Amazon FreeRTOS をダウンロードして設定する	90
Amazon FreeRTOS デモプロジェクトを構築および実行する	91
トラブルシューティング	91
NXP LPC54018 IoT モジュール	91
環境をセットアップする	91
Amazon FreeRTOS デモプロジェクトを構築および実行する	92
トラブルシューティング	94
Renesas Starter Kit+ for RX65N-2MB	94
Renesas ハードウェアのセットアップ	94
環境をセットアップする	94
Amazon FreeRTOS サンプルをビルドおよび実行する	95
トラブルシューティング	99
STMicroelectronics STM32L4 Discovery Kit IoT Node	99
環境をセットアップする	99
Amazon FreeRTOS デモプロジェクトを構築および実行する	99
トラブルシューティング	101
Texas Instruments CC3220SF-LAUNCHXL	101
環境をセットアップする	101
Amazon FreeRTOS デモプロジェクトを構築および実行する	103
トラブルシューティング	104
Windows Device Simulator	105
環境をセットアップする	105
Amazon FreeRTOS デモプロジェクトを構築および実行する	106
トラブルシューティング	106
Xilinx Avnet MicroZed Industrial IoT キット	106
MicroZed ハードウェアのセットアップ	106
環境をセットアップする	107
Amazon FreeRTOS デモプロジェクトを構築および実行する	109
トラブルシューティング	118
Amazon FreeRTOS ライブドリ	119
Amazon FreeRTOS 移植ライブドリ	119
Amazon FreeRTOS アプリケーションライブドリ	131
AWS IoT Device Defender	140
概要	140
ソースとヘッダーファイル	140
開発者サポート	141
Amazon FreeRTOS Device Defender API	141
使用例	143
AWS IoT デバイスシャドウ	144
概要	144
依存関係と要件	144
ソースとヘッダーファイル	145
API リファレンス	145
使用例	145
AWS IoT Greengrass	146
概要	146

依存関係と要件	146
ソースとヘッダーファイル	146
API リファレンス	147
使用例	147
Bluetooth Low Energy	148
概要	148
依存関係と要件	149
機能	150
ソースとヘッダーファイル	150
Amazon FreeRTOSBLE ライブラリ設定ファイル	151
最適化	151
使用制限	151
初期化	152
API リファレンス	153
使用例	153
移植	155
Amazon FreeRTOS Bluetooth デバイス用の Mobile SDK	157
MQTT (ペータ)	158
概要	158
依存関係と要件	159
機能	159
設定	159
API リファレンス	160
使用例	160
MQTT (レガシー)	161
概要	161
依存関係と要件	161
機能	161
ソースとヘッダーファイル	162
主な設定	162
最適化	163
開発者サポート	163
初期化	164
API リファレンス	164
移植	164
無線通信 (OTA) エージェント	164
概要	164
機能	165
ソースとヘッダーファイル	165
API リファレンス	165
使用例	166
移植	166
公開鍵暗号標準 (PKCS) #11	166
概要	166
機能	166
非対称暗号化方式のサポート	167
セキュアソケット	168
概要	168
依存関係と要件	169
機能	169
フットプリント	170
ソースとヘッダーファイル	170
トラブルシューティング	170
開発者サポート	170
使用制限	171
初期化	171
API リファレンス	171

使用例	171
移植	173
Transport Layer Security (TLS)	173
Wi-Fi	173
概要	173
依存関係と要件	173
機能	173
フットプリント	175
ソースとヘッダーファイル	175
設定	175
初期化	176
API リファレンス	176
使用例	176
移植	178
Amazon FreeRTOS デモ	179
Amazon FreeRTOS デモを実行する	179
デモを設定する	179
AWS IoT Greengrass	179
AWS IoT デバイスシャドウ	181
Bluetooth Low Energy	183
概要	183
前提条件	183
共通コンポーネント	185
MQTT over BLE	189
Wi-Fi プロビジョニング	190
汎用属性サーバー	193
Microchip Curiosity PIC32MZEF 用のブートローダー	194
ブートローダーのステート	194
フラッシュデバイス	195
アプリケーションイメージ構造	195
イメージヘッダー	196
イメージ記述子	197
イメージトレーラー	198
ブートローダーの設定	198
ブートローダーの構築	199
無線による更新	199
セキュアソケット	202
デバイスの移植	204
Bootloader	204
ログ記録	204
ログ作成設定	204
接続	205
Wi-Fi 管理	205
ソケット	205
セキュリティ	206
TLS	206
PKCS #11	207
Amazon FreeRTOS のカスタムライブラリを使用する	208
OTA ポータブル抽象化レイヤー	208
デバイスに資格を与える	210
AWS IoT Device Tester for Amazon FreeRTOS を使用する	211
AWS IoT Device Tester for Amazon FreeRTOS バージョン	211
前提条件	212
Amazon FreeRTOS をダウンロードする	212
AWS IoT Device Tester for Amazon FreeRTOS をダウンロードする	213
AWS アカウントを作成して設定する	213
AWS コマンドラインインターフェイス (CLI) をインストールする	213

マイクロコントローラー ボードの適格性確認を初めて行う	214
ライブラリポートイングレイヤーを追加する	214
AWS 認証情報を設定する	214
AWS IoT Device Tester でデバイスプールを作成する	215
ビルド、フラッシュ、テストを設定する	218
Amazon FreeRTOS 適格性確認スイートの実行	223
AWS IoT Device Tester コマンド	224
結果とログ	224
結果の表示	224
適格性再確認のためのテスト	226
トラブルシューティング	226
デバイス設定のトラブルシューティング	227
アクセス許可ポリシーテンプレート	230

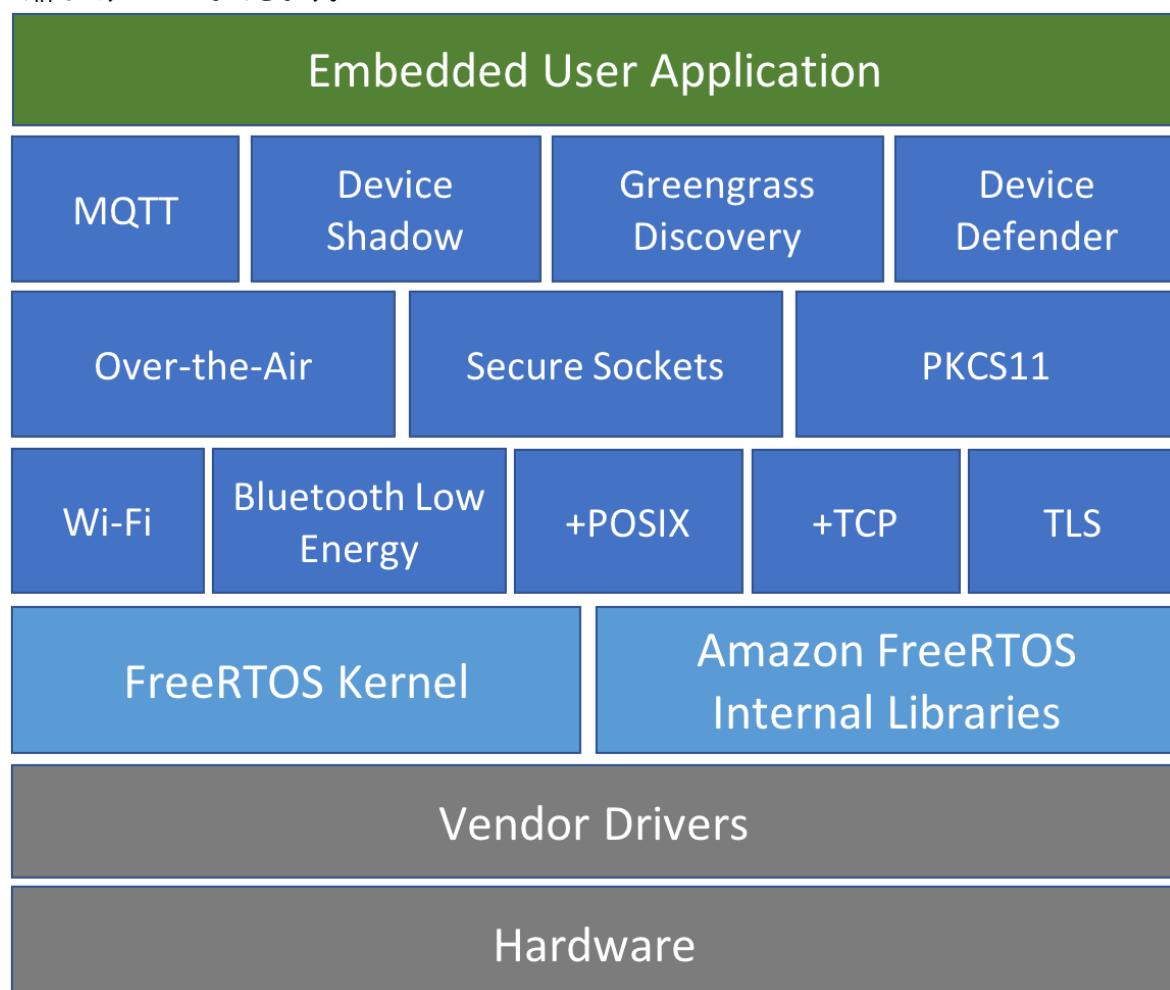
Amazon FreeRTOS とは。

Amazon FreeRTOS は、FreeRTOS カーネルに接続性、セキュリティ、無線 (OTA) 用のライブラリを追加したリアルタイムオペレーティングシステムです。Amazon FreeRTOS には、適格なボード上で Amazon FreeRTOS 機能を実演するデモアプリケーションも含まれています。

Amazon FreeRTOS アーキテクチャ

Amazon FreeRTOS は通常、デバイスのアプリケーションに必要なすべてのコンポーネントを含む、単一のコンパイル済みイメージとしてデバイスにフラッシュされます。このイメージは、組み込み開発者が作成したアプリケーション、Amazon が提供するソフトウェアライブラリ、FreeRTOS カーネル、およびハードウェアプラットフォーム用のドライバーとボードサポートパッケージ (BSP) の機能を組み合わせたものです。使用されている個々のマイクロコントローラとは別に、組み込みアプリケーション開発者は、FreeRTOS カーネルおよびすべての Amazon FreeRTOS ソフトウェアライブラリと同じ標準化されたインターフェースを期待できます。

下の図のアイコンをクリックして、Amazon FreeRTOS アーキテクチャーの特定のコンポーネントについて詳しく知ることができます。



FreeRTOS カーネル

FreeRTOS カーネルは、多数のアーキテクチャをサポートするリアルタイムオペレーティングシステム カーネルであり、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。カーネルでは以下のサポートが提供されています。

- 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(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。

FreeRTOS カーネルの詳細については、「[FreeRTOS カーネルの基礎 \(p. 2\)](#)」を参照してください。

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。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。

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。これらの機能は、タスク間およびタスク間のシグナリングシナリオの大部分で使用できます。

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。

メモリ管理

このセクションでは、カーネルのメモリ割り当てとアプリケーションのメモリ管理について説明します。

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することができます。

- 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管理

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ができますが、フリーブロックに隣接するメモリは結合しません。

`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。

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、トーカ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。

タスクへのダイレクト通知

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る、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。ストリームバッファ機能は、プロジェクトに `<BASE_DIR>/libs/FreeRTOS/stream_buffer.c` ソースファイルを含めることで有効になります。

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、ロック時間の指定が行えます。ストリームバッファへ書き込むロック時間が 0 以外で `xStreamBufferSend()` が呼び出され、かつバッファがいっぱいの場合、スペースが使用可能になるかロック時間が切れるまで、タスクは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。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` に独自の `sbSEND_COMPLETED()` を実装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、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、またはロック時間が切れるまで、タスクはロックされた状態になります。

タスクが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、タスクのロック時間が切れると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ロックされたタスクが、バッファ内で使用可能になるよう待機しているかを確認し、その場合はロック状態からタスクを削除します。`FreeRTOSConfig.h` に代替の実装を提供することによって、`sbRECEIVE_COMPLETED()` デフォルトの動作を変更できます。

メッセージバッファ

メッセージバッファは、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。例えば、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の上に構築されます。メッセージバッファ機能は、プロジェクトに `<BASE_DIR>/libs/FreeRTOS/stream_buffer.c` ソースファイルを含めることで有効になります。

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に、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` に独自の `sbSEND_COMPLETED()` を実装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ブロックされたタスクが、バッファ内で使用可能になるよう待機しているかを確認し、その場合はブロック状態からタスクを削除します。`FreeRTOSConfig.h` に代替の実装を提供することによって、`sbRECEIVE_COMPLETED()` デフォルトの動作を変更できます。

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。タイマーの開始から、実行されるコールバック関数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、ティックごとに低電力状態の開始/終了がなされるため、最も軽い電力節約モードを除く、すべての省電力モードを上回る電力および時間が消費されてしまいます。

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させる時まで、マイクロコントローラはディープパワーセービング状態になります。

Amazon FreeRTOS ライブラリ

Amazon FreeRTOS には、以下を可能にするライブラリが含まれています。

- ・MQTT と Device Shadows 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。
- ・Amazon FreeRTOS 無線による更新 (p. 7) をリッスンして処理します。

詳細については、「[Amazon FreeRTOS ライブラリ](#)」を参照してください。

Amazon FreeRTOS 無線による更新

OTA(無線通信経由)更新を使用すると、スポット群の 1 つまたは複数のデバイスにファイルを展開できます。OTA 更新はデバイスマーケットのアップデートに使用するよう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。ファイルを無線経由で送信するときは、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、ファイル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 更新マネージャサービス \(p. 45\)](#) 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、デバイス上で動作し AWS IoT ジョブの通知トピックに登録して更新メッセージを待ち受ける Amazon 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 AWS IoT ストリーミングトピックにリクエストをパブリッシュし、MQTT プロトコルを使用してファイルロックを受信します。ロックをファイルに再アセンブルし、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ます。ファイルが有効な場合、ファームウェアの更新がインストールされます。Amazon 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、[Amazon FreeRTOS 無線通信 \(OTA\) エージェントライブラリ \(p. 164\)](#) 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。

Amazon FreeRTOS 無線による更新により、次のことが可能になります。

- 展開する前に、ファームウェアにデジタル署名し、暗号化します。
- 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gingします。

無線による更新の前提条件

無線による更新を使用するには、次の操作を行う必要があります。

- [更新を保存する Amazon S3 バケットを作成する \(p. 8\)](#).
- [OTA 更新サービスロールの作成 \(p. 9\)](#).
- [OTA ユーザーポリシーの作成 \(p. 10\)](#).
- [コード署名証明書の作成 \(p. 12\)](#).
- Code Signing for AWS IoT を使用している場合は、[Code Signing for AWS IoT へのアクセスの許可 \(p. 17\)](#)。
- [OTA ライブラリで Amazon FreeRTOS をダウンロードする \(p. 17\)](#).Amazon FreeRTOS を使用していない場合は、独自の OTA エージェントの実装を提供できます。

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 AWS IoT を使用している場合、コード署名ジョブを作成するため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) および出力先バケット (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) を使用します。ソースと出力先の両方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. <https://console.aws.amazon.com/s3/> に移動します。
2. [Create bucket] を選択します。
3. バケット名を入力し、[Next (次へ)] を選択します。
4. [/バージョニング] で、すべてのバージョンが同じバケットに保持されていることを確認し、[次へ] を選択します。

5. [Next (次へ)] を選択して、デフォルトのアクセス許可を受け入れます。
6. [Create bucket] を選択します。

Amazon S3 の詳細については、「[Amazon Simple Storage Service Console User Guide](#)」を参照してください。

OTA 更新サービスロールの作成

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: Tags (次の手順: タグ)] を選択します。
8. [Next: Review] を選択します。
9. ロールの名前と説明を入力し、[Create role (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。

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. [Attach policies (ポリシーをアタッチします)] 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から [AmazonFreeRTOSOTAUpdate] を確認して、[ポリシーのアタッチ] を選択します。サービスロールにポリシーがアタッチされます。
4. 前のステップと同じ手順に従って、**AWSIoTTTRegistration** ポリシーをサービスロールに添付します。

IAM サービスロールに必要なアクセス権限を追加するには

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] 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>"  
    }]  
}
```

必ず `<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ます。

5. [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。

Amazon S3 サービスロールに必要なアクセス権限を追加するには

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] 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"  
            ],  
            "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>` を、お使いのバケットの名前に置き換えてください。

5. [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

IAM 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。IAM 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。
- AWS Certificate Manager に保存された証明書にアクセスする。
- AWS IoT ストリーミングサービスにアクセスする。
- Amazon FreeRTOS OTA 更新にアクセスする。
- AWS IoT ジョブにアクセスする。
- IAM にアクセスする。
- Code Signing for AWS IoT にアクセスする。
- Amazon FreeRTOS ハードウェアプラットフォームを一覧表示する。

IAM ユーザーに必要なアクセス許可を与えるには、OTA ユーザーポリシーを作成し、それを IAM ユーザーにアタッチします。詳細については、[IAM ポリシー](#)を参照してください。

OTA ユーザー policy を作成するには

1. <https://console.aws.amazon.com/iam/> コンソールを開きます。
2. ナビゲーションペインで [Users] を選択します。
3. リストから IAM ユーザーを選択します。
4. [Add permissions] を選択します。
5. [Attach existing policies directly] を選択します。
6. [Create policy] を選択します。
7. [JSON] タブを選択し、次の policy 文書をコピーして policy エディタに貼り付けます。

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

<example-bucket> を OTA 更新ファームウェアイメージが保存されている Amazon S3 バケットの名前に置き換えます。<your-account-id> は自分の AWS アカウント ID に置き換えます。AWS アカウント ID は、コンソールの右上で確認できます。アカウント ID を入力する際、ダッシュ (-) をすべて削除します。<role-name> を、作成したばかりの IAM サービスロールの名前に置き換えます。

8. [ポリシーの確認] を選択します。
9. 新しい OTA ユーザー policy 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 policy を IAM ユーザーにアタッチするには

1. IAM コンソールのナビゲーションペインで [Users (ユーザー)] を選択し、ユーザーを選択します。
2. [Add permissions] を選択します。

3. [Attach existing policies directly] を選択します。
4. 作成した OTA ユーザーポリシーを検索し、その横にあるチェックボックスをオンにします。
5. [Next: Review] を選択します。
6. [Add permissions] を選択します。

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。実稼働環境では、よく知られている認証機関 (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 Amazon 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 の CC3200SF-LAUNCHXL のコード署名証明書の作成 \(p. 12\)](#)
- [Microchip Curiosity PIC32MZEF のコード署名証明書の作成 \(p. 14\)](#)
- [Espressif ESP32 のコード署名証明書の作成 \(p. 15\)](#)
- [Amazon FreeRTOS Windows Simulator のコード署名証明書の作成 \(p. 16\)](#)
- [カスタムハードウェアのコード署名証明書の作成 \(p. 16\)](#)

Texas Instruments の CC3200SF-LAUNCHXL のコード署名証明書の作成

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で OTA の更新を試すことができます。

コード署名証明書、プライベートキー、および証明書チェーンを AWS Certificate Manager にインポートするには、AWS Command Line Interface を使用する必要があります。AWS CLI のインストールおよび使用の詳細については、「[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 と Code Signing for AWS IoT の基準を満たすコード署名証明書を作成するには、以下のステップに従ってください。

Note

コード署名証明書を作成するには、コンピュータに OpenSSL をインストールする必要があります。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

- 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。
- 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
```

- 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
```

- 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
```

- 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
```

- 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、および証明書チェーンを AWS Certificate Manager にインポートします。

```
aws acm import-certificate --certificate file://tisigner.crt.pem --private-key file://tisigner.key --certificate-chain file://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 AWS IoT 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に書かれています。Code Signing for AWS IoT を使用することをお勧めしますが、手動でファームウェアイメージに署名することもできます。

Microchip Curiosity PIC32MZEF のコード署名証明書の作成

Microchip Curiosity PIC32MZEF 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。

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、および証明書チェーンを AWS Certificate Manager にインポートします。

```
aws acm import-certificate --certificate file://ecdsasigner.crt --private-key  
file://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 AWS IoT 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に書かれています。Code Signing for AWS IoT を使用することをお勧めしますが、手動でファームウェアイメージに署名することもできます。

Espressif ESP32 のコード署名証明書の作成

Espressif ESP32 ボードは、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。

コード署名証明書、プライベートキー、および証明書チェーンを AWS Certificate Manager にインポートするには、AWS Command Line Interface を使用する必要があります。AWS CLI のインストールおよび使用の詳細については、「[AWS CLI のインストール](#)」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、および証明書チェーンを AWS Certificate Manager にインポートします。

```
aws acm import-certificate --certificate file://ecdsasigner.crt --private-key  
file://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 AWS IoT 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に書かれています。Code Signing for AWS IoT を使用することをお勧めしますが、手動でファームウェアイメージに署名することもできます。

Amazon FreeRTOS Windows Simulator のコード署名証明書の作成

Amazon 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、次のステップを使用して証明書を作成します。

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、および証明書チェーンを AWS Certificate Manager にインポートします。

```
aws acm import-certificate --certificate file://ecdsasigner.crt --private-key  
file://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 AWS IoT 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に書かれています。Code Signing for AWS IoT を使用することをお勧めしますが、手動でファームウェアイメージに署名することもできます。

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。

コード署名証明書、プライベートキー、および証明書チェーンを AWS Certificate Manager にインポートするには、AWS Command Line Interface を使用する必要があります。AWS CLI のインストールおよび使用の詳細については、「[AWS CLI のインストール](#)」を参照してください。

コード署名証明書を作成したのち、AWS CLI を使用して ACM にインポートできます。

```
aws acm import-certificate --certificate file://code-sign.crt --private-key file://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 Amazon FreeRTOS コードの一部である

`aws_ota_codesigner_certificate.h` ファイルに、コード署名証明書とプライベートキーの内容をコピー、貼り付け、フォーマットする必要があります。

Code Signing for AWS IoT へのアクセスの許可

本番環境では、ファームウェア更新プログラムのデジタル署名を行い、更新プログラムの信頼性と整合性を確保する必要があります。手動で更新プログラムに署名、または Code Signing for AWS IoT を使用してコードに署名することができます。Code Signing for Amazon FreeRTOS を使用するには、IAM ユーザーアカウントに Code Signing for Amazon FreeRTOS へのアクセスを許可する必要があります。

Code Signing for AWS IoT に IAM ユーザーアカウントのアクセス許可を付与するには

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、[Policies (ポリシー)] を選択します。
3. [ポリシーの作成] を選択します。
4. [JSON] タブで、次の JSON ドキュメントをコピーしてポリシーエディタに貼り付けます。このポリシーにより、IAM ユーザーはすべてのコード署名オペレーションにアクセスできます。

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

5. [ポリシーの確認] を選択します。
6. ポリシーネ名と説明を入力し、[Create policy (ポリシーの作成)] を選択します。
7. ナビゲーションペインで [Users] を選択します。
8. IAM ユーザーアカウントを選択します。
9. [Permissions] タブで、[Add permissions] を選択します。
10. [Attach existing policies directly (既存のポリシーを直接アタッチ)] を選択し、作成したコード署名ポリシーの横にあるチェックボックスをオンにします。
11. [Next: Review] を選択します。
12. [Add permissions] を選択します。

OTA ライブラリで Amazon FreeRTOS をダウンロードする

このセクションの手順に従い、コードをダウンロードしてデモアプリケーションを構築します。

Texas Instruments CC3200SF-LAUNCHXL 用の Amazon FreeRTOS のダウンロードと構築

Amazon FreeRTOS と OTA デモコードをダウンロードするには

1. AWS IoT コンソールを参照し、ナビゲーションペインから [Software (ソフトウェア)] を選択します。
2. [Amazon FreeRTOS Device Software (Amazon FreeRTOS デバイスソフトウェア)] の下で、[Configure download (ダウンロードの設定)] を選択します。
3. ソフトウェア設定のリストから、[Connect to AWS IoT - TI (AWS IoT - TI に接続)] を選択します。[Download (ダウンロード)] リンクではなく、設定名を選択します。
4. [Libraries (ライブラリ)] で [Add another library (別のライブラリを追加)] を選択し、[OTA Updates (OTA 更新)] を選択します。
5. [Demo Projects (デモプロジェクト)]、で [OTA Updates (OTA アップデート)] を選択します。
6. [Name required (名前が必要)] の下に **connect-to-IoT-OTA-TI** と入力し、[Create and download (作成してダウンロード)] を選択します。

Amazon FreeRTOS と OTA デモコードを含む zip ファイルをコンピュータに保存します。

デモアプリケーションを構築するには

1. .zip ファイルを展開します。
2. [Amazon FreeRTOS の使用開始 \(p. 59\)](#) の手順に従って、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 の SSID とパスワード、ボード用のプライベートキーと証明書を設定します。
3. Code Composer Studio でプロジェクトを開き、demos/common/demo_runner/aws_demo_runner.c を開きます。DEMO_RUNNER_RunDemos 関数を探し、vStartOTAUpdateDemoTask を除くすべての関数呼び出しがコメントアウトされていることを確認します。
4. 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。
5. 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
6. 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 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created
Device came up in Station mode
2 142 [Tmr Svc] Wi-Fi module initialized.
3 142 [Tmr Svc] Starting key provisioning...
4 142 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 340 [Tmr Svc] Write device certificate...
7 433 [Tmr Svc] Key provisioning done...
[WLAN EVENT] STA Connected to the AP: Mobile , BSSID: 24:de:c6:5d:32:a4
[NETAPP EVENT] IP acquired by the device

Device has connected to Mobile
Device IP Address is 192.168.111.12
```

```
8 2666 [Tmr Svc] Wi-Fi connected to AP Mobile.
9 2666 [Tmr Svc] IP Address acquired 192.168.111.12
10 2667 [OTA] OTA demo version 0.9.2
11 2667 [OTA] Creating MQTT Client...
12 2667 [OTA] Connecting to broker...
13 3512 [OTA] Connected to broker.
14 3715 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/OtaGA/jobs/$next/
get/accepted
15 4018 [OTA Task] [prvSubscribeToJobNotificationTopics] OK: $aws/things/OtaGA/jobs/notify-
next
16 4027 [OTA Task] [prvPAL_GetPlatformImageState] xFileInfo.Flags = 0250
17 4027 [OTA Task] [prvPAL_GetPlatformImageState] eOTA_PAL_ImageState_Valid
18 4034 [OTA Task] [OTA_CheckForUpdate] Request #0
19 4248 [OTA] [OTA_AgentInit] Ready.
20 4249 [OTA Task] [prvParseJSONbyModel] Extracted parameter [ clientToken: 0:OtaGA ]
21 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: execution
22 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: jobId
23 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: jobDocument
24 4249 [OTA Task] [prvParseJSONbyModel] parameter not present: afr_ota
25 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: streamname
26 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: files
27 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: filepath
28 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: filesize
29 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: fileid
30 4250 [OTA Task] [prvParseJSONbyModel] parameter not present: certfile
31 4251 [OTA Task] [prvParseJSONbyModel] parameter not present: sig-sha1-rsa
32 4251 [OTA Task] [prvParseJobDoc] Ignoring job without ID.
33 4251 [OTA Task] [prvOTA_Close] Context->0x2001b2c4
34 5248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
35 6248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
36 7248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
37 8248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
38 9248 [OTA] State: Ready Received: 1 Queued: 1 Processed: 1 Dropped: 0
```

Microchip Curiosity PIC32MZEF 用の Amazon FreeRTOS をダウンロードして構築する

Amazon FreeRTOS OTA デモコードをダウンロードするには

1. AWS IoT コンソールを参照し、ナビゲーションペインから [Software (ソフトウェア)] を選択します。
2. [Amazon FreeRTOS Device Software (Amazon FreeRTOS デバイスソフトウェア)] の下で、[Configure download (ダウンロードの設定)] を選択します。
3. ソフトウェア設定のリストから、[Connect to AWS IoT - Microchip (AWS IoT - Microchip に接続)] を選択します。[Download (ダウンロード)] リンクではなく、設定名を選択します。
4. [Libraries (ライブラリ)] で [Add another library (別のライブラリを追加)] を選択し、[OTA Updates (OTA 更新)] を選択します。
5. [Demo projects (デモプロジェクト)] の下で、[OTA Update (OTA アップデート)] を選択します。
6. [Name required (名前が必要)] で、カスタム Amazon FreeRTOS ソフトウェア設定の名前を入力します。
7. [Create and download (作成してダウンロード)] を選択します。

OTA 更新デモアプリケーションを構築するには

1. ダウンロードした .zip ファイルを展開します。
2. [Amazon FreeRTOS の使用開始 \(p. 59\)](#) の手順に従って、aws_demos プロジェクトを MPLAB X IDE にインポートし、AWS IoT エンドポイント、Wi-Fi の SSID とパスワード、ボード用のプライベートキーと証明書を設定します。

3. オープン aws_demos/lib/aws/ota/aws_ota_codesigner_certificate.h.
4. コード署名証明書の内容を static const char signingcredentialSIGNING_CERTIFICATE_PEM 変数に貼り付けます。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" "FnRlc3Rf621nbmVyQGFtYXpvi5jb20wHhcNMTcxMTAzMTkxDOM1WhcNMTgxMTAz\n" "MTkxDOM2WjAhMR8wHQYDVQBZZ0ZXN0X3NpZ25lckBhbWF6b24uY29tMFkwEwYH\n" "KoZIZj0CAQYIKoZIZj0DAQcDQgAERavZfvwL1X+E4dIF7dbkVMUu4IrJ1CAsFkc8\n" "gZxPzn683H40XMK1tDZPEwr9ng78w+QYQg7ygnr2stz8yhh06MkMCiwcWYDVROP\n" "BAQDAgeAMBMDGA1UdJQQMMAoGCCsGAQUFBwMDMAoGCCqGSM49BAMCA0gAMEUCIF0R\n" "r5cb7rEUNtW0vGd05Macrg0AbfSoVYvBOK9fp63WAqt5h3BaS123coKSGg84twlq\n" "TkO/pV/xEmyZmZdV+HxV/OM=\n" "-----END CERTIFICATE-----\n";
```

5. Python 3 以降をインストールします。
6. pip install pyopenssl を実行し、pyOpenSSL をインストールします。
7. \demos\common\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。
8. MPLAB X IDE でプロジェクトを開き、demos/common/demo_runner/aws_demo_runner.c を開きます。DEMO_RUNNER_RunDemos 関数を探し、vStartOTAUpdateDemoTask を除くすべての関数呼び出しがコメントアウトされていることを確認します。
9. 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。
10. 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。
 - ポーレート: 115200
 - データビット: 8
 - パリティ: なし
 - ストップビット: 1
11. ボードからデバッガーを切り離し、ボード上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てください。

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] タブを選択し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] vDHCPPProcess: offer ac140a0eip
```

```
1 36297 [IP-task] vDHCPPProcess: 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] vDHCPPProcess: offer c0a804beip
3 7230 [IP-task] vDHCPPProcess: 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] MOTT 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 AWS IoT に接続し、OTA 更新に必要な MQTT トピックに登録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 用の Amazon FreeRTOS をダウンロードして構築する

1. Amazon FreeRTOS のソースを [GitHub](#) からダウンロードしてください。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. テキストエディタで `demos/common/demo_runner/aws_demo_runner.c` を開きます。`DEMO_RUNNER_RunDemos` 関数を探し、`vStartOTAUpdateDemoTask` を除くすべての関数呼び出しがコメントアウトされていることを確認します。
4. `demos/espressif/esp32_devkitc_esp_wrover_kit/make` 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](#)」の説明に従って、デモプログラムをフラッシュし、`make flash monitor` を実行してその出力を確認することができます。
5. OTA 更新デモを実行する前に以下を確認してください。
 - `vStartOTAUpdateDemoTask` が、`demos/common/demo_runner/aws_demo_runner.c` の `DEMO_RUNNER_RunDemos()` 関数で呼び出される唯一の関数であることを確認してください。
 - SHA-256/ECDSA コード署名証明書が `demos/common/include/aws_ota_codesigner_certificate.h` にコピーされていることを確認してください。

カスタムハードウェアプラットフォーム用の Amazon FreeRTOS をダウンロードして構築する

Amazon FreeRTOS のソースを [GitHub](#) からダウンロードしてください。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。

プロジェクトを構築して実行し、AWS IoT に接続できることを確認します。

Amazon FreeRTOS を新しいプラットフォームに移植する方法については、「[デバイスの移植 \(p. 204\)](#)」を参照してください。

OTA チュートリアル

このセクションでは、OTA 更新を使用して Amazon 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について説明します。ただし、OTA 更新を使用して AWS IoT に接続されているデバイスにファイルを送信することもできます。

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 更新ジョブの自動化、多数のデバイスの操作、および Amazon FreeRTOS に適さないデバイスの使用の際に役立ちます。Amazon FreeRTOS でのデバイスの認定の詳細については、[Amazon FreeRTOS の「パートナー」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 AWS IoT を使用している場合は、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。

デバイス上の Amazon 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 Amazon 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。Amazon 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 CC3200SF-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 \(p. 24\)](#)
- [Microchip Curiosity PIC32MZEF 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 \(p. 27\)](#)
- [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 \(p. 29\)](#)
- [Windows Simulator の初期ファームウェア \(p. 31\)](#)
- [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 \(p. 31\)](#)

Texas Instruments CC3200SF-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

これらの手順は、[Texas Instruments CC3200SF-LAUNCHXL 用の Amazon FreeRTOS のダウンロードと構築 \(p. 18\)](#) 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 CC3200SF-LAUNCHXL のピンの中央のセット (位置 = 1) に SOP ジャンパを配置し、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 を起動し、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 (デバイスマード)] で、[Develop (開発)] を選択します。[Create Project (プロジェクトの作成)] を選択します。
6. ターミナルエミュレータを切断します。
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 (ファイルを追加する)] アイコンを選択します .
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 (ファイルを追加する)] アイコンを選択します .
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 AmazonFreeRTOS/demos/ti/cc3200_launchpad/ccs/Debug ディレクトリにあります。[Open] を選択します。
 - 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、[Browse (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
```

Microchip Curiosity PIC32MZEF 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

これらの手順は、[Microchip Curiosity PIC32MZEF 用の Amazon FreeRTOS をダウンロードして構築する \(p. 19\)](#) 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。

デモアプリケーションをボードに焼き付けるには

1. aws_demos プロジェクトを再構築し、エラーなしでコンパイルすることを確認してください。
2.  ツールバーで [] を選択します。
3. プログラミングプロセスが完了したら、ICD 4 デバッガを取り外してボードをリセットします。ターミナルエミュレータを再接続して、MPLAB X IDE 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。

ターミナルは、次のような出力を表示する必要があります。

```
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] vDHCPPProcess: offer ac140a0eip
                                         1 36297 [IP-task] vDHCPPProcess: 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

```

次の手順では、参照ブートローダと暗号署名付きアプリケーションで構成される統合された 16 進ファイルまたはファクトリイメージを作成します。ブートローダは、ブート時にアプリケーションの暗号署名を検証し、OTA の更新をサポートします。

ファクトリイメージを構築してフラッシュするには

1. [Source Forge](#) から SRecord ツールがインストールされていることを確認してください。srec_cat および srec_info プログラムを含むディレクトリがシステムパスにあることを確認します。
2. ファクトリイメージの OTA シーケンス番号およびアプリケーションバージョンを更新します。
3. aws_demos プロジェクトを構築します。
4. factory_image_generator.py スクリプトを実行して、ファクトリイメージを生成します。

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

このコマンドでは、以下のパラメータを使用します。

- mplab.production.bin: アプリケーションバイナリ
- MCHP-Curiosity-PIC32MZEF: プラットフォーム名
- private_key.pem: コード署名プライベートキー
- aws_bootloader.X.production.hex: ブートローダ 16 進数ファイル

aws_demos プロジェクトを構築すると、アプリケーションのバイナリイメージとブートローダの 16 進数ファイルがプロセスの一部として構築されます。demos/microchip/ ディレクトリ下の各プロジェクトには、これらのファイルを含む dist/pic32mz_ef_curiosity/

production/ ディレクトリが含まれています。生成された統合 16 進数ファイルの名前は `mplab.production.unified.factory.hex` です。

5. 生成された 16 進ファイルをデバイスにプログラムするには、MPLab IPE ツールを使用します。
6. イメージがアップロードされると、ボードの UART 出力を見ることでファクトリイメージが機能しているかどうかを確認することができます。すべてが正しく設定されていれば、イメージが正常に起動します。

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

7. 証明書が正しく設定されていない場合、または OTA イメージが正しく署名されていない場合、チップのブートローダが無効なアップデートを消去する前に、次のようなメッセージが表示されることがあります。コード署名証明書が一貫していることを確認し、前の手順を慎重に確認してください。

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

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

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インが正しく設定されていることを確認するために「[Amazon FreeRTOS の使用を開始する](#)」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. `demos/common/demo_runner/aws_demo_runner.c` の `DEMO_RUNNER_RunDemos` 関数で、`vStartOTAUpdateDemoTask` を除くすべての関数の呼び出しをコメントアウトします。
2. 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
BLKO of EFUSE
I (123) system_api: Base MAC address is not set, read default base MAC address from
BLKO 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
[ ... ]
```

- 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。[Ctrl+J] を押すと終了します。お好みの 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ができます。ボードのシリアルポートに接続すると再起動が始まることもあることに注意してください。

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。Amazon FreeRTOS のソースコードの構造の詳細については詳細については、「[Amazon FreeRTOS デモ \(p. 179\)](#)」を参照してください。

Amazon 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

Amazon 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。

- IDE で aws_demos プロジェクトを開きます。

2. `demos/common/include/aws_application_version.h` を開き、`APP_VERSION_BUILD` トークン値を増やします。
3. Microchip Curiosity PIC32MZEF を使用している場合は、`\demos\common\ota\bootloader\utility\user-config\ota-descriptor.config` にある OTA シーケンス番号を増やしてください。新しい OTA イメージが生成されるたびに、OTA シーケンス番号を増やす必要があります。
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 \(p. 8\)](#) 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 CC3200SF-LAUNCHXL: `demos\ti\cc3220_launchpad\ccs\debug\aws_demos.bin`
- Microchip Curiosity PIC32MZEF: `demos\microchip\curiosity_pic32mzef\mplab\dist\pic32mz_ef_curiosity\production\mplab.production.ota.bin`
- Espressif ESP32: `demos\espressif\esp32_devkitc_esp_wrover_kit\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で、[Manage (管理)]、[Jobs (ジョブ)] の順に選択します。
2. [Create (作成)] を選択します。
3. [Create an Amazon FreeRTOS Over-the-Air (OTA) update job (Amazon FreeRTOS OTA (無線通信経由) 更新ジョブの作成)] の下で、[Create OTA update job (OTA 更新ジョブの作成)] を選択します。
4. 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。[Select devices to update (更新するデバイスの選択)] の下で、[Select (選択)] を選択します。デバイスのグループを更新するには、[Things (モノ)] タブを選択します。デバイスのグループを更新するには、[Thing Groups (モノのグループ)] タブを選択します。
5. 単一のデバイスを更新する場合は、デバイスに関連付けられている IoT モノの横にあるチェックボックスをオンにします。デバイスのグループを更新する場合は、デバイスに関連付けられているモノのグループの横にあるチェックボックスをオンにします。[次へ] を選択します。
6. [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。
7. [Code signing profile (コード署名プロファイル)] の下で、[Create (作成)] を選択します。
8. [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

Amazon 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 以外の暗号化スイートを使用している場合は、Code Signing for AWS IoT を使用することも、ファームウェア更新に自分で署名することもできます。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 \(p. 35\)](#)」を参照してください。

- b. [Code signing certificate (コード署名証明書)] で、[Select (選択)] を選択して以前にインポートした証明書を選択するか、[Import (インポート)] をクリック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。証明書の場所はプラットフォームによって異なり、[初期ファームウェアのインストール \(p. 24\)](#) の指示に従ったときにコード署名証明書を配置した場所になります。

Note

Microchip Curiosity PIC32MZEF で実行する場合、コード署名証明書は最初に証明書ストア内で名前で検索されます。見つからない場合は、組み込みの証明書が使用されます。

Important

このプラットフォーム上の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。

9. [Select your firmware image in S3 or upload it (S3 のファームウェアイメージを選択またはアップロードします)] で、[Select (選択)] 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。

10. [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、ファームウェアイメージがコピーされるデバイスの場所の完全修飾パス名を入力します。この場所も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。

11. [IAM role for OTA update job (OTA 更新ジョブの IAM ロール)] の下で、S3 バケットへのアクセスを許可するロールを選択し、次のポリシーを設定します。
 - AWSIoTThingsRegistration
 - AmazonFreeRTOSOTAUpdate
12. [次へ] を選択します。
13. [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。
14. OTA 更新ジョブの ID と説明を入力し、[Create (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、ファームウェアイメージがコピーされるデバイスの場所の完全修飾パス名を入力します。これはプラットフォームによって異なる場合があります。
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

- [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。
- [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。これはプラットフォームによって異なる場合があります。
- [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)] の下に、ファームウェアイメージがコピーされるデバイスの場所の完全修飾パス名を入力します。これはプラットフォームによって異なる場合があります。
- [Signature (署名)] で、PEM 形式の署名を貼り付けます。
- [Original hash algorithm (元のハッシュアルゴリズム)] の下で、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。
- [Original encryption algorithm (元の暗号化アルゴリズム)] の下で、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。
- [Select your firmware image in Amazon S3 (Amazon S3 のファームウェアイメージの選択)] の下で、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には次のようなものがあります。

- お名前
- IP アドレス
- E メールアドレス
- お客様の場所
- 銀行の情報
- 医療情報

- [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。
- [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。
- 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、[IN PROGRESS] というステータスで 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は更新しません。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の詳細は、「[無線による更新デモアプリケーション \(p. 199\)](#)」を参照してください。

AWS CLI を使用して OTA 更新を作成する

AWS CLI を使用して OTA 更新を作成する手順は次のとおりです。

1. フ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

CLI を使用して OTA の更新を実行する場合は、Code Signing for AWS IoT を使用するか、ファームウェア更新に署名することができます。Code Signing for AWS IoT によってサポートされている暗号化署名およびハッシュアルゴリズムのリストについては、[SigningConfigurationOverrides](#) を参照してください。Code Signing for AWS IoT 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、それを Amazon S3 にアップロードする前にファームウェアのバイナリに署名する必要があります。

Code Signing for Amazon FreeRTOS を使用したファームウェアイメージへの署名

Code Signing for AWS IoT を使用してファームウェアイメージに署名するには、[コード署名ツール](#)をインストールする必要があります。インストール手順については、ツールをダウンロードし、README ファイルをお読みください。Code Signing for AWS IoT の詳細については「[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 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 バケットにすることができます。

次のテキストでは、start-signing-job コマンドのパラメータについて説明します。

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。Amazon S3 コンソールを参照し、バケットを選択し、ページの上部にある [Versions (バージョン)] の横にある [Show (表示)] を選択すると表示することができます。

destination

署名付きファームウェアの S3 バケット内の送信先。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。

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。生成されたファイル名は、Amazon S3 コンソールを参照し、バケットを選択することで見つけることができます。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 AWS IoT の詳細については、「[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

OTA 更新サービスは、MQTT メッセージの更新を送信します。これを行うには、署名済みファームウェアの更新を含むストリームを作成する必要があります。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>  
    }  
  }  
]
```

次のリストでは、JSON ファイルの属性について説明します。

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 AWS IoT を使用している場合は、ファイル名は Code Signing for AWS IoT によって生成された GUID です。

`create-stream` CLI コマンドを使用して、ストリームを作成します。

```
aws iot create-stream --stream-id <your_stream_id> --description <your_description> --files file://<stream.json> --role-arn <your_role_arn>
```

次のリストは、`create-stream` CLI コマンドの引数について説明しています。

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 AWS IoT を使用する場合は、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

Amazon S3 バケットへのアクセスを許可する IAM ロール

署名済ファームウェアイメージの Amazon S3 オブジェクトキーを検索するには、aws signer describe-signing-job --job-id <my-job-id> コマンドを使用します。my-job-id は、create-signing-job CLI コマンドで表示されるジョブ 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

OTA 更新ジョブを作成するには、create-ota-update CLI コマンドを使用します。

```
aws iot create-ota-update --ota-update-id "<my_ota_update>" --target-selection SNAPSHOT
--description "<a cli ota update>" --files file://<ota.json> --targets arn:aws:iot:<your-
aws-region>:<your-aws-account>:thing/<your-thing-name> --role-arn arn:aws:iam::<your-aws-
account>:role/<your-ota-service-role>
```

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には次のようなものがあります。

- お名前
- IP アドレス
- E メールアドレス
- お客様の場所
- 銀行の情報
- 医療情報

ota-update-id

任意の OTA 更新 ID。

target-selection

有効な値は次のとおりです。

- SNAPSHOT: 選択した IoT モノに更新をデプロイした後、ジョブは終了します。
- CONTINUOUS: ジョブは、選択したグループに追加されたデバイスへの更新を引き続きデプロイします。

description

OTA 更新の説明テキスト。

files

OTA 更新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、次の属性が含めることができます。

- fileName: 完全修飾ファームウェアイメージファイル名。Texas Instruments CC3200SF-LAUNCHXL の場合、この値は "/sys/mcuflashimg.bin" を指定する必要があります。Microchip の場合、この値は "mplab.production.bin" を指定する必要があります。

- `fileLocation`: ファームウェアの更新に関する情報が含まれています。
 - `stream`: ファームウェアの更新を含むストリーム。
 - `streamId`: `create-stream` CLI コマンドで指定されたストリーム ID。
 - `fileId`: `create-stream` に渡された JSON ファイルで指定されたファイル ID。
 - `s3Location`: Amazon S3 のファームウェア更新の場所。
 - `bucket`: ファームウェア更新を含む Amazon S3 バケット。
 - `key`: ファームウェア更新キー。
 - `version`: ファームウェア更新のバージョン。
- `codeSigning`: コード署名ジョブに関する情報が含まれています。
 - `awsSignerJobId`: `start-signing-job` コマンドによって生成された署名者ジョブ ID。
 - `startSigningJobParamater`: コード署名ジョブを開始するために必要な情報。
 - `signingProfileParameter`: 署名ジョブプロファイルの作成に必要な情報。
 - `certificateArn`: コード署名ジョブの作成に使用された証明書の ACM ARN。
 - `platformId`: 使用しているハードウェアプラットフォームの ID。
 - `certificatePathOnDevice`: デバイスの証明書へのパス。
 - `signingProfileName`: コード署名プロファイルの名前。この名前のプロファイルが存在しない場合は、`signingProfileParameter` の値を指定する必要があります。指定された名前のプロファイルが存在し、`signingProfileParameter` の値を指定する場合は、指定した値と署名プロファイルに使用した値が正確に一致している必要があります。
 - `destination`: 署名付きアーティファクトが配置される場所。
 - `s3Destination`: 署名付きアーティファクトが配置される Amazon S3 バケット。
 - `bucket`: Amazon S3 バケット。
 - `prefix`: コード署名アーティファクトのプレフィックス。デフォルトでは、`signedImage/` です。これにより、フォルダの下に `signedImage` という名前のフォルダが作成されます。
 - `customCodeSigning`: カスタム署名に関する情報が含まれています。
 - `signature`: カスタム署名が含まれています。
 - `inlineDocument`: カスタム署名。
 - `certificateChain`: カスタム署名の証明書チェーンが含まれています。
 - `certificateName`: デバイスのコード署名証明書のパス名。
 - `inlineDocument`: 証明書チェーン。
 - `hashAlgorithm`: 署名を作成するために使用されるハッシュアルゴリズム。
 - `signatureAlgorithm`: コード署名に使用される署名アルゴリズム。
 - `attributes`: 任意のキーと値のペア。

`targets`

OTA 更新で更新するデバイスを指定する 1 つ以上の IoT モノ ARN。

`role-arn`

サービスロールの ARN。

Code Signing for AWS IoT を使用する `create-ota-update` というコマンドに渡される JSON ファイルの例を次に示します。

```
[  
 {  
   "fileName": "firmware.bin",  
   "fileLocation": {  
     "stream": {  
       "streamId": "004",  
     }  
   }  
 }]
```

```
        "fileId":123
    },
},
"codeSigning": {
    "awsSignerJobId": "48c67f3c-63bb-4f92-a98a-4ee0fb2bef6"
}
]
]
```

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` という CLI コマンドに渡される JSON ファイルの例を次に示します。

```
[
{
    "fileName": "firmware.bin",
    "fileLocation": {
        "stream": {
            "streamId": "004",
            "fileId": 123
        }
    },
    "codeSigning": {
        "customCodeSigning": {
            "signature": {
                "inlineDocument": "<your_signature>"
            },
            "certificateChain": {
                "certificateName": "<your_certificate_name>",
                "inlineDocument": "<your_certificate_chain>"
            },
            "hashAlgorithm": "<your_hash_algorithm>",
            "signatureAlgorithm": "<your_signature_algorithm>"
        }
    }
}]
```

Amazon 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およびコード署名のストリームを作成できるようにする `create-ota-update` という CLI コマンドに渡される JSON ファイルの例を次に示します。

```
[
{
    "fileName": "<your_firmware_path_on_device>",
    "fileVersion": "1",
    "fileLocation": {
        "s3Location": {
            "bucket": "<your_bucket_name>",
            "key": "<your_object_key>",
            "version": "<your_S3_object_version>"
        }
    },
    "codeSigning": {
        "startSigningJobParameter": {
            "signingProfileName": "myTestProfile",
            "signingProfileParameter": {
                "certificateArn": "<your_certificate_arn>",
                "platformId": "<your_platform_id>",
                "certificatePathOnDevice": "<certificate_path>"
            }
        },
        "destination": {
            "s3Destination": {
                "bucket": "<your_destination_bucket>"
            }
        }
    }
}]
```

```
        }
    }
}
]
```

既存のプロファイルを使用して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` という CLI コマンドに渡される JSON ファイルの例を次に示します。

```
[
{
  "fileName": "<your_firmware_path_on_device>",
  "fileVersion": "1",
  "fileLocation": {
    "s3Location": {
      "bucket": "<your_bucket_name>",
      "key": "<your_object_key>",
      "version": "<your_S3_object_version>"
    }
  },
  "codeSigning": {
    "startSigningJobParameter": {
      "signingProfileName": "<your_unique_profile_name>",
      "destination": {
        "s3Destination": {
          "bucket": "<our_destination_bucket>"
        }
      }
    }
  }
}]
```

Amazon FreeRTOS OTA が既存のコード署名ジョブ ID を持つストリームを作成できるようにする `create-ota-update` という CLI コマンドに渡される JSON ファイルの例を次に示します。

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

OTA 更新を作成する `create-ota-update` という CLI コマンドに渡される JSON ファイルの例を次に示します。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>",
  "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

すべての OTA 更新のリストを取得するには、list-ota-updates という CLI コマンドを使用できます。

```
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 CLI コマンドを使用して、OTA 更新の作成または削除のステータスを取得できます。

```
aws iot get-ota-update --ota-update-id <your-ota-update-id>
```

コマンド get-ota-update の出力は次のようにになります。

```
{  
    "otaUpdateInfo": {  
        "otaUpdateId": "myotaupdate1",  
        "otaUpdateArn":  
            "arn:aws:iot:us-west-2:123456789012:otaupdate/my_ota_update",  
        "creationDate": 1522444438.424,  
        "lastModifiedDate": 1522444440.681,  
    }  
}
```

```
"description": "a test OTA update",
"targets": [
    "arn:aws:iot:us-west-2:123456789012:thing/myDevice"
],
"targetSelection": "SNAPSHOT",
"otaUpdateFiles": [
    {
        "fileName": "app.bin",
        "fileLocation": {
            "stream": {
                "streamId": "003",
                "fileId": 123
            }
        }
    }
],
"codeSigning": {
    "awsSignerJobId": "592932bb-24a1-4f91-8ddd-66145352ad19",
    "customCodeSigning": {}
}
},
"otaUpdateStatus": "OTA-STATUS",
"awsIotJobId": "f76da3c0_10eb_41df_9029_ba7abc20f609",
"awsIotJobArn": "arn:aws:iot:us-
west-2:123456789012:job/f76da3c0_10eb_41df_9029_ba7abc20f609"
}
```

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場合は、`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

現時点では、AWS IoT コンソールを使用してストリーム、または OTA 更新を削除することはできません。AWS CLI を使用して、ストリーム、OTA 更新、および OTA 更新中に作成された IoT ジョブを削除できます。

OTA ストリームを削除する

OTA 更新を作成時に、ユーザーまたは AWS IoT コンソールは、ファームウェアをチャンクに分割するストリームを作成し、MQTT を介して送信できます。このストリームは、`delete-stream` CLI コマンドで削除できます。以下に例を示します。

```
aws iot delete-stream --stream-id <your_stream_id>
```

OTA ストリームの削除

OTA 更新を作成すると、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。
- 更新される各デバイスの 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

Amazon 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` という CLI コマンドを使用して、ジョブおよび関連するジョブの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。ジョブの削除中、そのジョブのステータスは DELETION_IN_PROGRESS として表示されます。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、1つのデバイスのジョブ実行が削除されます。以下に例を示します。

```
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 CLI コマンドの場合と同様に、delete-job-execution に --force パラメータを渡して強制的に実行ジョブの実行を削除することができます。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、「[無線による更新デモアプリケーション \(p. 199\)](#)」を参照してください。

OTA 更新マネージャサービス

OTA 更新マネージャサービスで以下のことができます。

- OTA 更新を作成します。
- 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。
- ターゲットデバイスに送信するファイルのリスト。
- AWS IoT ジョブサービス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 更新はデバイスマームウェアの更新に使用するよう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。ファイルを無線経由で送信するときは、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、ファイルにデジタル署名することをお勧めします。「[Code Signing for Amazon FreeRTOS](#)」を使用してファイルに署名することも、独自のコード署名ツールを使用することもできます。

ファイルにデジタル署名された後、Amazon ストリーミングサービスを使用してストリームを作成します。このサービスは、ファイルを MQTT を介してデバイスに送信できるブロックに分割します。

OTA 更新を作成すると、OTA マネージャ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。Amazon FreeRTOS OTA エージェントはデバイス上で動作し、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、MQTT を介して更新をストリーミングし、ファイルをローカルに保管します。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

ムウェアの更新をインストールします。Amazon 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、更新をリッシュしてダウンロードし、インストール操作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

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、オプションで、OTA 完了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。

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理解する必要があります。AWS IoT 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、および OTA 更新ジョブで AWS アカウントを設定する方法については、「[Amazon FreeRTOS の前提条件](#)」を参照してください。

MQTT 接続管理

OTA エージェントは、AWS IoT サービスとのすべての通信に 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。

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 完了コールバックを使用し、1 秒ごとに統計情報をいくつか出力するだけの設定をします。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。

AWS IoT MQTT ブローカーを使用する実例については、「[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。

```
/* Create the MQTT Client. */
if( MQTT_AGENT_Create( &( xMQTT_h ) ) == eMQTTAgentSuccess )
{
    for ( ; ; )
    {
        memset( &xConnParm, 0, sizeof( xConnParm ) );

        /* ... Set MQTT connection parameters here per your application needs ... */

        configPRINTF( ( "Connecting to %s\r\n", clientcredentialMQTT_BROKER_ENDPOINT ) );
        if( MQTT_AGENT_Connect( xMQTT_h, &( xConnParm ),
myappMAX_AWS_CONNECT_WAIT_IN_TICKS ) == eMQTTAgentSuccess )
        {
            configPRINTF( ( "Connected to broker.\r\n" ) );

            /* Initialize the OTA Agent with the default completion callback handler. */
            OTA_AgentInit( xMQTT_h, ( const uint8_t * )( clientcredentialIOT_THING_NAME ), NULL,
/* NULL uses the default
callback handler. */ ( TickType_t ) ~0 );

            while( ( eState = OTA_GetAgentState() ) != eOTA_AgentState_NotReady )
            {
                /* Wait forever for OTA traffic but allow other tasks to run
and output statistics only once per second. */

                vTaskDelay( myappONE_SECOND_DELAY_IN_TICKS );
            }
        }
    }
}
```

```

        configPRINTF( ( "State: %s Received: %u Queued: %u Processed: %u
Dropped: %u\r\n",
                pcStateStr[eState],
                OTA_GetPacketsReceived(),
                OTA_GetPacketsQueued(),
                OTA_GetPacketsProcessed(),
                OTA_GetPacketsDropped() ) );
    }
    /* ... Handle MQTT disconnect per your application needs ... */
}
else
{
    configPRINTF( ( "ERROR: MQTT_AGENT_Connect() Failed.\r\n" ) );
}
/* After failure to connect or a disconnect, wait an arbitrary one second before
retry. */
vTaskDelay( myappONE_SECOND_DELAY_IN_TICKS );
}
else
{
    configPRINTF( ( "Failed to create MQTT client.\r\n" ) );
}

```

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 回の統計出力をループします。
- エージェントが停止した場合は、1 秒待ってから再度接続してください。

OTA 完了イベントのカスタムコールバックの使用

前の例では、OTA_AgentInit API の 3 番目のパラメータに NULL を指定し、OTA 完了イベントの組み込みコールバックandler を使用しました。完了イベントのカスタム処理を実装する場合は、コールバックandler の関数アドレスを OTA_AgentInit API に渡す必要があります。OTA プロセス中、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andler 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。

```

/**
 * @brief OTA Job callback events.
 *
 * After an OTA update image is received and authenticated, the agent calls the user
 * callback (set with the OTA_AgentInit API) with the value eOTA_JobEvent_Activate to
 * signal that the device must be rebooted to activate the new image. When the device
 * boots, if the OTA job status is in self test mode, the agent calls the user callback
 * with the value eOTA_JobEvent_StartTest, signaling that any additional self tests
 * should be performed.
 *
 * If the OTA receive fails for any reason, the agent calls the user callback with
 * the value eOTA_JobEvent_Fail instead to allow the user to log the failure and take
 * any action deemed appropriate by the user code.
 */
typedef enum {
    eOTA_JobEvent_Activate, /*! OTA receive is authenticated and ready to activate. */
    eOTA_JobEvent_Fail,     /*! OTA receive failed. Unable to use this update. */
    eOTA_JobEvent_StartTest /*! OTA job is now in self test, perform user tests. */
} OTA_JobEvent_t;

```

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なります。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ジョブイベントです。

eOTA_JobEvent_Activate イベント

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を延期することができます。

eOTA_JobEvent_Fail イベント

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。

eOTA_JobEvent_StartTest イベント

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ことを判断する前に、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、OTA エージェントはテストの準備ができ次第 eOTA_JobEvent_StartTest 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、必要と思われるテストの追加を選択できます。デバイスファームウェアがセルフテストによって信頼できると判断された場合、コードは OTA_SetImageState(eOTA_ImageState_Accepted) 関数を呼び出すことによってファームウェアを新しい永続的なイメージとしてコミットする必要があります。

デバイスに特別なハードウェアやテストが必要なメカニズムがない場合は、デフォルトのコールバックハンドラーを使用できます。eOTA_JobEvent_Activate イベントを受信すると、デフォルトのハンドラーがデバイスを直ちにリセットします。

OTA セキュリティ

以下に示しているのは、OTA セキュリティの 3 つの側面です。

接続の安全性

OTA 更新マネージャーは、AWS IoT で使用される 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 更新のトラフィックは、AWS IoT デバイスゲートウェイを通して、AWS IoT セキュリティメカニズムを使用します。デバイスゲートウェイを介した各送受信 MQTT メッセージは、厳密な認証と許可を受けます。

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。Amazon FreeRTOS OTA 更新マネージャーは、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 の認証と許可を受けます。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、ストリーミング中に Amazon S3 に保存されたファームウェア更新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のアクセス許可を与える必要があります。

Code Signing for AWS IoT

AWS IoT コンソールは [Code Signing for AWS IoT](#) を使用し、AWS IoT 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。

Code Signing for AWS IoT は、ACM にインポートする証明書とプライベートキーを使用します。テスト用に自己署名証明書を使用できますが、よく知られている商用認証機関 (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 (キーの使用状況)] と [Extended Key Usage (拡張キー用途)] の拡張機能を使用します。[Key Usage (キーの使用状況)] 拡張機能は Digital Signature に設定され、[Extended Key Usage (拡張キー用途)] 拡張機能は Code Signing に設定されます。コードイメージへの署名の詳細については、[Code Signing for AWS IoT 開発者ガイド](#)および[Code Signing for AWS IoT API リファレンス](#)を参照してください。

Note

Code Signing for AWS IoT SDK のコード署名は <https://tools.signer.aws.a2z.com/awssigner-tools-v2.zip> から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 の設定 \(p. 49\)](#)
- [AWS CloudTrail での AWS IoT OTA API 呼び出しのログ作成 \(p. 53\)](#)
- [Texas Instruments CC3220SF Launchpad を使用した OTA 更新とトラブルシューティング \(p. 55\)](#)

OTA 更新のための CloudWatch Logs の設定

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、Amazon CloudWatch ログ記録を有効にして設定し、OTA 更新を設定することができます。CloudWatch Logs の詳細については、「[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 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することができます。詳細については、「[Amazon CloudWatch Logs に対する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、ログ記録用のロールを作成し、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)] の下で、[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. [Update] 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 ジョブが作成される。
- AWS IoT ジョブが完了する。
- 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. [Log Groups (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 です。

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: ストリームの要求が AWS IoT ストリーミングサービスに送信されました。
- DescribeStream: ストリームに関する情報の要求が AWS IoT ストリーミングサービスに送信されました。

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追加情報。

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。

プロトコル

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。

status

ログを生成したオペレーションのステータス。有効な値は次のとおりです。

- Success
- 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": "875157236366",  
    "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."
}
```

デバイスがストリーミングサービスからストリームを要求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-4ed1-9b07-286afed7e4c8streams/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ロ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."
}
```

AWS CloudTrail での AWS IoT OTA API 呼び出しのログ作成

Amazon FreeRTOS は、すべての AWS IoT OTA API 呼び出しをキャプチャ、および指定した Amazon S3 バケットにログファイルを記録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 AWS IoT OTA API への API 呼び出しをキャプチャします。CloudTrail によって収集された情報を使用して、リクエストの作成元のソース IP アドレス、リクエストの実行者、リクエストの実行日時など、AWS IoT OTA に対してどのようなリクエストが行われたかを判断することができます。

CloudTrail の詳細 (設定して有効にする方法など) については、[AWS CloudTrail User Guide](#) を参照してください。

CloudTrail 内の Amazon FreeRTOS 情報

AWS アカウントで CloudTrail ログ記録を有効にすると、AWS IoT OTA アクションに対するほとんどの API コールは CloudTrail ログファイルに記録されます。この際、他の AWS サービスのレコードと一緒にログファイルに書き込まれます。CloudTrail は、期間とファイルサイズに基づいて新規ファイルの作成と書き込みのタイミングを決定します。

Note

AWS IoT OTA データプレーンのアクション (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には CloudWatch を使用します。

AWS IoT OTA のコントロールプレーンは、CloudTrail によってログに記録されます。たとえば、CreateOTAUpdate、GetOTAUpdate、CreateStream セクションの呼び出しが、CloudTrail ログファイルにエントリーを生成します。

各ログエントリには、誰がリクエストを生成したかに関する情報が含まれます。ログエントリのユーザー ID 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、ルートと IAM ユーザー認証情報のどちらを使用して送信されたか。
- リクエストが、ロールとフェデレーティッドユーザーのどちらの一時的なセキュリティ認証情報を使用して送信されたか。
- リクエストが、別の AWS サービスによって送信されたかどうか。

詳細については、「[CloudTrail userIdentity 要素](#)」を参照してください。AWS OTA IoT のアクションは [AWS IoT OTA API リファレンス](#) で説明されています。

必要な場合はログファイルを自身の Amazon S3 バケットに保管できますが、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、新しいログファイルの配信時に 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 の Amazon SNS 通知の設定](#)」を参照してください。

複数の AWS リージョンと複数の AWS アカウントから単一の Amazon S3 バケットに AWS IoT OTA ログファイルを集めることもできます。

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「[複数のアカウントから CloudTrail ログファイルを受け取る](#)」を参照してください。

Amazon FreeRTOS ログファイルエントリの概要

CloudTrail ログファイルには、1つ以上のログエントリを含めることができます。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ー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 呼び出し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 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>"
}
```

Texas Instruments CC3220SF Launchpad を使用した OTA 更新とトラブルシューティング

CC3220SF Launchpad プラットフォームは、セキュリティアラートカウンターを使用するソフトウェアの改ざん検出メカニズムを提供します。セキュリティアラートカウンターは、整合性違反があるたびに 1 ずつ増えています。セキュリティアラートカウンターが事前に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。ロックされたデバイスに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。

Amazon FreeRTOS ソースコードのダウンロード

[Amazon FreeRTOS コンソール](#) から、Amazon FreeRTOS で認定されたプラットフォーム用に設定された Amazon FreeRTOS のバージョンをダウンロードできます。認定されたプラットフォームの一覧については、[Amazon FreeRTOS パートナー](#) ウェブサイトを参照してください。

Amazon FreeRTOS は [GitHub](#) からダウンロードすることもできます。

Amazon FreeRTOS コンソール

[Amazon FreeRTOS コンソール](#) から、マイクロコントローラーベースのデバイス用のアプリケーションを作成するために必要なものがすべて含まれたパッケージの設定およびダウンロードができます。

- FreeRTOS カーネル
- Amazon FreeRTOS ライブラリ
- プラットフォームサポートライブラリ
- ハードウェアドライバー

Amazon FreeRTOS コンソールの詳細については、「[Amazon FreeRTOS コンソール \(p. 55\)](#)」を参照してください。

Amazon FreeRTOS コンソール

[Amazon FreeRTOS コンソール](#) では、事前定義された設定でパッケージをダウンロードすることができ、またはアプリケーションに必要なハードウェアプラットフォームとライブラリを選択して、独自の設定を作成できます。これらの設定は AWS に保存され、いつでもダウンロードできます。

事前定義された Amazon FreeRTOS 構成

事前定義された設定により、必要なライブラリについて検討することなく、サポートされているユースケースを素早く開始することができます。事前定義された設定を使用するには、「[Amazon FreeRTOS コンソール](#)」を参照し、使用する設定を検索し、[Download (ダウンロード)] を選択します。

Amazon FreeRTOS のバージョン、ハードウェアプラットフォーム、または設定のライブラリを変更する場合は、事前定義された設定をカスタマイズすることもできます。事前定義された設定をカスタマイズすると、新しいカスタム設定が作成され、Amazon FreeRTOS コンソールの事前定義された設定は上書きされません。

事前定義された設定からカスタム設定を作成するには

1. 「[Amazon FreeRTOS コンソール](#)」を参照します。
2. ナビゲーションペインで、[Software (ソフトウェア)] を選択します。
3. [Amazon FreeRTOS Device Software (Amazon FreeRTOS デバイスソフトウェア)] の下で、[Configure download (ダウンロードの設定)] を選択します。
4. カスタマイズする事前定義された設定の横にある省略記号を選択し、[Customize (カスタマイズ)] を選択します。
5. [Configure Amazon FreeRTOS Software (Amazon FreeRTOS ソフトウェアの設定)] ページで、Amazon FreeRTOS のバージョン、ハードウェアプラットフォーム、およびライブラリを選択し、新しい設定に名前と説明を付けます。
6. ページの下部の [Create and download (作成してダウンロード)] を選択して、カスタム設定を作成してダウンロードします。

カスタム Amazon FreeRTOS 設定

カスタム設定では、ハードウェアプラットフォーム、統合開発プラットフォーム (IDE)、コンパイラ、および必要な RTOS ライブラリのみを指定できます。これにより、デバイス内にアプリケーションコードのためのスペースが得られます。

カスタム設定を作成するには

1. 「[Amazon FreeRTOS コンソール](#)」を参照し、[Create new (新規作成)] を選択します。
2. 使用する Amazon FreeRTOS のバージョンを選択します。最新バージョンがデフォルトで使用されます。
3. [New Software Configuration (新しいソフトウェア設定)] ページで、[Select a hardware platform (ハードウェアプラットフォームを選択)] を選択し、事前準備されたプラットフォームの 1 つを選択します。
4. 使用する IDE およびコンパイラを選択します。
5. 必要な Amazon FreeRTOS ライブラリの場合は、[Add Library (ライブラリの追加)] を選択します。別のライブラリが必要としているライブラリを選択した場合は、そのライブラリが追加されます。他のライブラリを選択する場合は、[Add another library (別のライブラリを追加)] を選択します。
6. [Demo Projects (デモプロジェクト)] セクションで、デモプロジェクトの 1 つを有効にします。これにより、プロジェクトファイルのデモが有効になります。
7. [Name required (名前が必要)] で、カスタム設定の名前を入力します。

Note

カスタム設定の名前に個人を特定できる情報を使用しないでください。

8. [Description (説明)] に、カスタム設定の説明を入力します。
9. ページの下部の [Create and download (作成してダウンロード)] を選択して、カスタム設定を作成してダウンロードします。

カスタム設定を編集するには

1. 「[Amazon FreeRTOS コンソール](#)」を参照します。
2. ナビゲーションペインで、[Software (ソフトウェア)] を選択します。

3. [Amazon FreeRTOS Device Software (Amazon FreeRTOS デバイスソフトウェア)] の下で、[Configure download (ダウンロードの設定)] を選択します。
4. 編集する設定の横にある省略記号を選択し、[Edit (編集)] を選択します。
5. [Configure Amazon FreeRTOS Software (Amazon FreeRTOS ソフトウェアの設定)] ページで、設定の Amazon FreeRTOS のバージョン、ハードウェアプラットフォーム、およびライブラリを変更できます。
6. ページの下部の [Save and download (保存してダウンロード)] を選択して、設定を保存してダウンロードします。

カスタム設定を削除するには

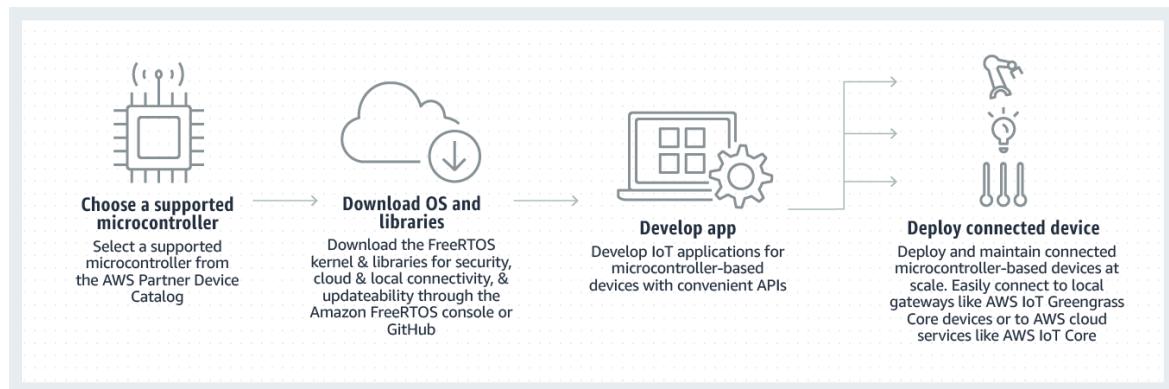
1. 「[Amazon FreeRTOS コンソール](#)」を参照します。
2. ナビゲーションペインで、[Software (ソフトウェア)] を選択します。
3. [Amazon FreeRTOS Device Software (Amazon FreeRTOS デバイスソフトウェア)] の下で、[Configure download (ダウンロードの設定)] を選択します。
4. 削除する設定の横にある省略記号を選択し、[Delete (削除)] を選択します。

クイック接続ワークフロー

Amazon FreeRTOS コンソールには、設定が事前定義されているすべてのボード用のクイック接続ワークフローも含まれます。クイック接続ワークフローは、AWS IoT および AWS IoT Greengrass 向けの Amazon FreeRTOS デモアプリケーションを設定して実行するのに役立ちます。開始するには、[Predefined configurations (事前定義された設定)] タブを選択し、ボードを検索して [Quick connect (クイック接続)] を選択し、クイック接続ワークフローのステップに従います。

開発ワークフロー

開発を開始するには、Amazon 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、完全な IoT ソリューションの一部として AWS IoT サービスまたは AWS IoT Greengrass に接続できます。



その他のリソース

AWS または Amazon FreeRTOS についてその他のご質問がある場合は、以下のリソースが役立つと思われます。

リソース	説明
GitHub の Amazon FreeRTOS	Amazon FreeRTOS について Amazon FreeRTOS エンジニアリングチームに確認したいご質問がある場合は、Amazon FreeRTOS の GitHub ページで問題を提起してください。
AWS フォーラム	AWS や Amazon FreeRTOS に関する技術的な質問について AWS コミュニティでディスカッションするには、フォーラムにアクセスしてください。
AWS サポートセンター	AWS のテクニカルサポートを受けるには、サポートセンターにアクセスしてください。
お問い合わせ	AWS の請求、アカウントサービス、イベント、不正使用、および AWS に関するその他の問題についてお問い合わせの場合には、[お問い合わせ] ページにアクセスしてください。

Amazon FreeRTOS の使用開始

Amazon FreeRTOS の開始方法チュートリアルは、ホストマシンで Amazon 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 コンソールについて理解していることを前提としています。続行する前に、[AWS IoT の開始方法](#)のチュートリアルを完了していない場合は、完了することをお勧めします。

最初のステップ

開始するには、[最初のステップ \(p. 60\)](#) を参照してください。

ボード固有の入門ガイド

[最初のステップ \(p. 60\)](#) を完了した後、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、その後でボードでデモをコンパイルして実行できます。ボード固有の手順については、ボードの入門ガイドを参照してください。

- Cypress CYW943907AEVAL1F 開発キットの開始方法 (p. 67)
- Cypress CYW954907AEVAL1F 開発キットの開始方法 (p. 68)
- Infineon XMC4800 IoT 接続キットの開始方法 (p. 79)
- MediaTek MT7697Hx Development Kit の開始方法 (p. 82)
- Microchip Curiosity PIC32MZEF の開始方法 (p. 85)
- NXP LPC54018 IoT モジュールの開始方法 (p. 91)
- Renesas Starter Kit+ for RX65N-2MB の開始方法 (p. 94)
- STMicroelectronics STM32L4 ディスカバリキット IoT ノード用の開始方法 (p. 99)
- Texas Instruments CC3220SF-LAUNCHXL の開始方法 (p. 101)
- Windows Device Simulator の開始方法 (p. 105)
- Xilinx Avnet MicroZed Industrial IoT キットの開始方法 (p. 106)

Note

次の自己完結型 Amazon FreeRTOS の開始方法ガイドの最初のステップを完了する必要はありません。

- Espressif ESP32-DevKitC と ESP-WROVER-KIT の開始方法 (p. 69)
- Nordic nRF52840-DK の開始方法 (p. 88)

トラブルシューティング

開始の際に発生した問題の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 \(p. 66\)](#) を参照してください。ボード固有のトラブルシューティングのヒントについては、ボードの入門ガイドを参照してください。

Amazon FreeRTOS - 認定済み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 Amazon FreeRTOS に適合しています。

- Cypress CYW943907AEVAL1F Development Kit
- Cypress CYW954907AEVAL1F Development Kit
- Espressif ESP32-DevKitC
- Espressif ESP-WROVER-KIT
- Infineon XMC4800 IoT Connectivity Kit
- MediaTek MT7697Hx Development Kit
- Microchip Curiosity PIC32MZEF / バンドル
- Nordic nRF52840-DK (ベータ)
- NXP LPC54018 IoT モジュール
- Renesas RX65N RSK IoT モジュール
- STMicroelectronics STM32L4 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。

新しいデバイスの認定については、「[デバイスに資格を与える \(p. 210\)](#)」を参照してください。

最初のステップ

Amazon FreeRTOS を開始するには、AWS アカウント、AWS IoT および Amazon FreeRTOS クラウドサービスにアクセスするアクセス許可を持つ IAM ユーザー、およびサポートされているハードウェアプラットフォームのいずれかが必要です。また、Amazon FreeRTOS をダウンロードして、AWS IoT と連携するようにボードの Amazon FreeRTOS デモプロジェクトを設定する必要があります。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。

1. AWS アカウントとアクセス許可の設定 (p. 61)

「[AWS アカウントとアクセス許可の設定 \(p. 61\)](#)」の手順を完了したら、[Amazon FreeRTOS コンソール](#)の [Quick Connect (クイック接続)] ワークフローに従ってボードを AWS クラウドにすばやく接続できます。[Quick Connect (クイック接続)] のワークフローに従っている場合は、このリストの残りの手順を完了する必要はありません。以下のボードでは、Amazon FreeRTOS の設定は現在 Amazon FreeRTOS コンソールでは使用できません。

- Cypress CYW943907AEVAL1F 開発キット
- Cypress CYW954907AEVAL1F 開発キット
- Espressif ESP-WROVER-KIT
- Espressif ESP32-DevKitC
- Nordic nRF52840-DK

2. AWS IoT で MCU ボードを登録 (p. 61)

3. Amazon FreeRTOS のダウンロード (p. 64)

4. Amazon FreeRTOS デモを設定する (p. 64)

Note

Espressif ESP32-DevKitC または ESP-WROVER-KIT 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 \(p. 69\)](#)」に進みます。

Nordic nRF52840-DK を使用している場合は、この手順をスキップして、「[Nordic nRF52840-DK の開始方法 \(p. 88\)](#)」に進みます。

AWS アカウントとアクセス許可の設定

AWS アカウントを作成するには、[Create and Activate an AWS Account (AWS アカウントの作成とアクティベーション)] を参照してください。

AWS アカウントに IAM ユーザーを追加するには、[IAM User Guide](#) を参照してください。AWS IoT および Amazon FreeRTOS に IAM ユーザーアカウントのアクセスを付与するには、IAM ユーザーアカウントに以下の IAM ポリシーをアタッチします。

- `AmazonFreeRTOSFullAccess`
- `AWSIoTFullAccess`

`AmazonFreeRTOSFullAccess` ポリシーを IAM ユーザーにアタッチするには

1. [IAM コンソール](#) を参照し、ナビゲーションペインから [Users (ユーザー)] を選択します。
2. 検索テキストボックスにユーザー名を入力し、リストから選択します。
3. [Add permissions] を選択します。
4. [Attach existing policies directly] を選択します。
5. 検索ボックスで「`AmazonFreeRTOSFullAccess`」と入力してリストから選択し、[Next: Review (次へ: レビュー)] を選択します。
6. [Add permissions] を選択します。

`AWSIoTFullAccess` ポリシーを IAM ユーザーにアタッチするには

1. [IAM コンソール](#) を参照し、ナビゲーションペインから [Users (ユーザー)] を選択します。
2. 検索テキストボックスにユーザー名を入力し、リストから選択します。
3. [Add permissions] を選択します。
4. [Attach existing policies directly] を選択します。
5. 検索ボックスで「`AWSIoTFullAccess`」と入力してリストから選択し、[Next: Review (次へ: レビュー)] を選択します。
6. [Add permissions] を選択します。

IAM およびユーザーアカウントの詳細については、[IAM User Guide](#) を参照してください。

ポリシーの詳細については、「[IAM アクセス権限とポリシー](#)」を参照してください。

AWS アカウントとアクセス許可を設定した後、[AWS IoT で MCU ボードを登録 \(p. 61\)](#) または [Amazon FreeRTOS コンソール](#) で [クイック接続] ワークフローを続行できます。

AWS IoT で MCU ボードを登録

AWS クラウドと通信するには、ボードを AWS IoT に登録する必要があります。ボードを AWS IoT に登録するには、以下が必要です。

AWS IoT ポリシー

AWS IoT ポリシーは、AWS IoT リソースへのアクセス権限をデバイスに付与します。これは AWS クラウド上に保存されます。

AWS IoT のモノ

AWS IoT のモノにより、AWS IoT でデバイスを管理できるようになります。これは AWS クラウド上に保存されます。

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、AWS IoT でデバイスを認証できるようになります。

Amazon FreeRTOS コンソールで [Quick Connect (クイック接続)] ワークフローを使用すると、ポリシー、AWS IoT のモノ、およびキーと証明書が自動的に作成されます。[Quick Connect (クイック接続)] ワークフローを使用している場合は、以下の手順を無視できます。

手動で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 アカウント番号を把握しておく必要があります。

AWS アカウント番号を見つけるには、AWS Management Console 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、[自分のアカウント] 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 リージョンを確認するには、AWS Command Line Interface を使用します。AWS CLI をインストールするには、AWS Command Line Interface ユーザーガイドの手順に従います。AWS CLI をインストールしたら、コマンドプロンプトウインドウを開き、次のコマンドを入力します。

```
aws iot describe-endpoint
```

出力は次のようにになります。

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

この例では、リージョンは us-west-2 です。

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インドウにコピーして貼り付けます。`aws-region` と `aws-account` をお客様の 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": "aws-region.amazonaws.com:8883"  
        }  
    ]  
}
```

```
        "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

AWS IoT メッセージプ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

freertos/demos/echo MQTT トピックで MQTT メッセージを発行するアクセス許可をデバイスに付与します。

iot:Subscribe

freertos/demos/echo MQTT トピックフィルターをサブスクライブするアクセス許可をデバイスに付与します。

iot:Receive

AWS IoT メッセージプローカーからメッセージを受信するアクセス許可をデバイスに付与します。

- [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

- 「[AWS IoT コンソール](#)」を参照します。
- 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。
- 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] を選択します。
- [Creating AWS IoT things (AWS IoT モノを作成する)] ページで、[Create a single thing (单一のモノを作成する)] を選択します。
- [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。
- [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。
- 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。
- 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、使用前にアクティベートする必要があります。

9. [Attach a policy (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権限を付与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 AWS IoT に登録されたら、「[Amazon FreeRTOS のダウンロード \(p. 64\)](#)」に進むことができます。

Amazon FreeRTOS のダウンロード

Amazon FreeRTOS は Amazon FreeRTOS コンソールまたは GitHub からダウンロードできます。

[Amazon FreeRTOS コンソール](#) の [Quick Connect (クイック接続)] ワークフローに従っている場合は、コンソールのワークフローのダウンロード手順に従い、次の手順は無視してください。

Amazon FreeRTOS コンソールから Amazon FreeRTOS をダウンロードするには

1. [Amazon FreeRTOS コンソール](#) に移動します。
2. [事前定義された設定] で、[Connect to AWS IoT - **Platform** (- プラットフォームに接続する)] を探し、次に [ダウンロード] を選択します。
3. ダウンロードしたファイルを AmazonFreeRTOS フォルダに解凍し、フォルダパスをメモします。

Note

Cypress CYW954907AEVAL1F または CYW943907AEVAL1F 開発キットを使い始めている場合は、GitHub から Amazon FreeRTOS をダウンロードする必要があります。これらのボードの Amazon FreeRTOS 設定は、現在 Amazon FreeRTOS コンソールからは使用できません。

GitHub から Amazon FreeRTOS をダウンロードするには

1. [Amazon FreeRTOS GitHub リポジトリ](#) にアクセスします。
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
```

4. amazon-freertos ディレクトリから、使用するブランチをチェックしてください。

Note

Microsoft Windows でファイルパスの最大長は 260 文字です。Amazon FreeRTOS のダウンロードディレクトリパスが長くなると、ビルドが失敗する可能性があります。
入門マニュアルでは、Amazon FreeRTOS ダウンロードディレクトリへのパスは `<BASE_FOLDER>` と呼ばれています。

Amazon FreeRTOS をダウンロードしたら、[Amazon FreeRTOS デモを設定する \(p. 64\)](#) に続行できます。

Amazon 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前に、Amazon 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。

[Amazon FreeRTOS コンソール](#) の [クイック接続] ワークフローに従っている場合は、コンソールのワークフローの設定手順に従い、以下の手順は無視してください。

AWS IoT エンドポイントを設定するには

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 [Settings] を選択します。

AWS IoT エンドポイントが [エンドポイント] に表示されます。次のように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 のモノ名が必要です。この名前を書き留めておきます。

4. オープン `<BASE_FOLDER>\demos\common\include\aws_clientcredential.h`
5. 以下の定数に値を指定します。

- static const char 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

Note

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. `<BASE_FOLDER>\demos\common\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 (オーブン、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 の認証情報をフォーマットするには

Note

AWS IoT 認証情報を設定するには、デバイスを登録する際に 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。AWS IoT のモノとしてデバイスを登録した後で、デバイスの証明書を AWS IoT コンソールから取得することはできますが、プライベートキーは取得できません。

Amazon 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、`<BASE_FOLDER>\tools\certificate_configuration\CertificateConfigurator.html` を開きます。
2. [Certificate PEM file (証明書 PEM ファイル)] の下で、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)] の下から、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 `<BASE_FOLDER>\demos\common\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。

Amazon FreeRTOS を設定した後、ボードの入門ガイドに進み、Amazon FreeRTOS デモをコンパイルして実行できます。この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、Hello World MQTT デモで、`<BASE_FOLDER>/demos/common/mqtt/aws_hello_world.c` にあります。

トラブルシューティングの開始方法

以下のトピックは、Amazon FreeRTOS の使用中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 (p. 66)
- ターミナルエミュレータをインストールする (p. 66)

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「[Amazon FreeRTOS の使用開始 \(p. 59\)](#)」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に AWS IoT コンソールにメッセージが表示されない場合は、次の操作を試してください。
 1. ターミナルウインドウを開き、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。

ターミナルエミュレータをインストールする

ターミナルエミュレータは、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があります。

ターミナルエミュレータ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

ターミナルの設定	値
データ	8 ビット
parity	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
```

Linux

```
ls /dev/tty*
```

macOS

```
ls /dev/cu.*
```

Cypress CYW943907AEVAL1F 開発キットの開始方法

Curiosity 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、MQTT Hello World 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 Amazon 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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

ボードにシリアル接続を設定したら、Amazon FreeRTOS デモプロジェクトをビルドし、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 Amazon FreeRTOS デモプロジェクトをビルドして実行するには

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。[全般] フォルダを展開し、[Existing Projects into Workspace (既存のプロジェクトを WorkSpace へ)] を選択してから、[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 <BASE_FOLDER>\demos\cypress\CYW954907AEVAL1F\wicedstudio を入力し、[開ける] を選択します。
4. [プロジェクト] の下で aws_demo プロジェクトが選択されている必要があります。[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 OFF)] 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。



Cypress CYW954907AEVAL1F 開発キットの開始方法

Curiosity 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、MQTT Hello World 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 Amazon 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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

ボードにシリアル接続を設定したら、Amazon FreeRTOS デモプロジェクトをビルドし、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。

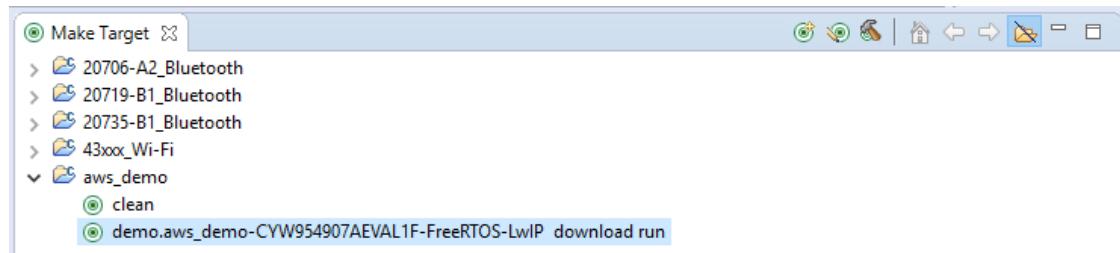
WICED Studio で Amazon FreeRTOS デモプロジェクトをビルドして実行するには

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。[全般] フォルダを展開し、[Existing Projects into Workspace (既存のプロジェクトを WorkSpace へ)] を選択してから、[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 <BASE_FOLDER>\demos\cypress\CYW954907AEVAL1F\wicedstudio を入力し、[開ける] を選択します。

4. [プロジェクト] の下で aws_demo プロジェクトが選択されている必要があります。[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 OFF)] 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。



Espressif ESP32-DevKitC と ESP-WROVER-KIT の開始方法

Espressif ESP32-DevKitCがない場合は、AWS Partner Device Catalog のパートナーから購入できます。Espressif ESP32-WROVER-KITがない場合は、AWS Partner Device Catalog のパートナーから購入できます。Amazon FreeRTOS では両方のデバイスがサポートされています。これらのボードの詳細については、Espressif ウェブサイトの「[ESP32-DevKitC](#)」または「[ESP-WROVER-KIT](#)」を参照してください。

Note

現在、ESP32-WROVER-KIT および ESP DevKitC の Amazon FreeRTOS ポートは、以下の機能をサポートしていません。

- Lightweight IP。
- 対称型マルチプロセッシング (SMP)。

前提条件

Espressif ボードで Amazon FreeRTOS を使い始める前に、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。

AWS アカウントを作成するには、「[AWS アカウントを作成し、有効化する](#)」を参照してください。

IAM ユーザーを AWS アカウントに追加するには、「[IAM User Guide](#)」を参照してください。AWS IoT および Amazon FreeRTOS に IAM ユーザーアカウントのアクセスを付与するには、IAM ユーザーアカウントに以下の IAM ポリシーをアタッチします。

- [AmazonFreeRTOSFullAccess](#)

- AWSIoTFullAccess

AmazonFreeRTOSFullAccess ポリシーを IAM ユーザーにアタッチするには

1. [IAM コンソール](#) を参照し、ナビゲーションペインから [Users (ユーザー)] を選択します。
2. 検索テキストボックスにユーザー名を入力し、リストから選択します。
3. [Add permissions (アクセス許可を追加する)] を選択します。
4. [Attach existing policies directly (既存のポリシーを直接アタッチする)] を選択します。
5. 検索ボックスで「**AmazonFreeRTOSFullAccess**」と入力してリストから選択し、[Next: Review (次へ: レビュー)] を選択します。
6. [Add permissions (アクセス許可を追加する)] を選択します。

AWSIoTFullAccess ポリシーを IAM ユーザーにアタッチするには

1. [IAM コンソール](#) を参照し、ナビゲーションペインから [Users (ユーザー)] を選択します。
2. 検索テキストボックスにユーザー名を入力し、リストから選択します。
3. [Add permissions (アクセス許可を追加する)] を選択します。
4. [Attach existing policies directly (既存のポリシーを直接アタッチする)] を選択します。
5. 検索ボックスで「**AWSIoTFullAccess**」と入力してリストから選択し、[Next: Review (次へ: レビュー)] を選択します。
6. [Add permissions (アクセス許可を追加する)] を選択します。

IAM およびユーザー アカウントの詳細については、[IAM User Guide](#) を参照してください。

ポリシーの詳細については、「[IAM アクセス権限とポリシー](#)」を参照してください。

Espressif ハードウェアのセットアップ[®]

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 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 入門ガイド](#) を参照してください。

Note

Espressif ガイドの「開始方法」に進まないでください。代わりに、次のステップを実行します。

環境をセットアップする

シリアル接続の確立

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、一部の 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。これらのドライバは [FTDI](#) からダウンロードできます。

詳細については、「[ESP32とのシリアル接続を確立する](#)」を参照してください。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。デモを構築する際に必要になります。

ツールチェーンの設定

ボードと通信するには、Espressif ツールチェーンを設定する必要があります。ツールチェーンを設定するには、ホストマシンのオペレーティングシステムの指示に従ってください。

Note

[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、停止してこのページの手順に戻ります。

以前に「ESP-IDF の取得」の指示に従って ESP-IDF をインストールした場合は、続行する前にシステムから `IDF_PATH` 環境変数を必ずクリアしてください。

- [Windows 用ツールチェーンの標準セットアップ](#)
- [macOS 用ツールチェーンの標準セットアップ](#)
- [Linux 用ツールチェーンの標準セットアップ](#)

Amazon FreeRTOS をダウンロードして設定する

環境をセットアップすると、GitHub から Amazon FreeRTOS をダウンロードできるようになります。Espressif ボード用の Amazon FreeRTOS の設定は、Amazon FreeRTOS コンソールからは利用できません。

Amazon FreeRTOS のダウンロード

[GitHub](#) から `amazon-freertos` リポジトリを複製またはダウンロードします。

Note

Microsoft Windows でのファイルパスの最大長は 260 文字です。Amazon FreeRTOS ダウンロードでの最長パスは 122 文字です。Amazon FreeRTOS プロジェクトのファイルに対応するため、AmazonFreeRTOS ディレクトリへのパスが 98 文字未満であることを確認してください。たとえば、`C:\Users\Username\Dev\AmazonFreeRTOS` の使用は可能ですが、`C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` というパスではビルドに失敗します。

このチュートリアルでは、ディレクトリへのパスを `AmazonFreeRTOS` 表記します。

Amazon FreeRTOS デモアプリケーションを設定する

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、`mingw32.exe` を開きます。
2. Python 2.7.10 以降がインストールされていることを確認するには、`python --version` を実行します。インストールされているバージョンが表示されます。Python 2.7.10 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 CLI が必要です。Windows を実行している場合は、`easy_install awscli` を使用して mingw32 環境に AWS 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 コマンドラインインターフェイスのインストール](#)」を参照してください。

4. `aws configure` を実行し、AWS アクセスキー ID、シークレットアクセスキー、およびデフォルトのリージョン名を使用して AWS 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 では、mingw32 環境で、easy_install boto3 を実行します。
- macOS または Linux で、pip install tornado nose --user、pip install boto3 --user の順に実行します。

Amazon 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、**<BASE_FOLDER>/tools/aws_config_quick_start/configure.json** を開いて次の属性を設定します。

afr_source_dir

コンピュータ上の amazon-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。

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)

設定スクリプトを実行するには

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、mingw32.exe を開きます。
2. **<BASE_FOLDER>/tools/aws_config_quick_start** ディレクトリに移動して、python SetupAWS.py setup を実行します。

スクリプトでは次のことが実行されます。

- 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 IoT エンドポイント、Wi-Fi SSID、および認証情報を aws_clientcredential.h 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.h ヘッダーファイルに書き込みます。

SetupAWS.py の詳細については、**<BASE_FOLDER>/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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

デモのフラッシュのためにボードの接続を設定するには

1. ホストコンピュータにボードを接続します。
2. macOS または Linux を使用している場合、ターミナルを開きます。Windows を使用している場合、mingw32.exe (msys ツールチェーンからダウンロードされたもの) を開きます。
3. <BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make に移動して次のコマンドを入力し、Espressif IoT 開発フレームワーク設定メニューを作成して開きます。

```
make menuconfig
```

メニューで選択を確定するには、[Select (選択)] を選択します。設定を保存するには、[Save (保存)] を選択します。メニューを終了するには、[Exit (終了)] を選択します。

4. Espressif IoT 開発フレームワーク設定メニューで、[Serial flasher config (シリアルフラッシャー設定)] に移動します。

シリアルポートを設定するには、[Default serial port (デフォルトのシリアルポート)] を選択します。ここで設定したシリアルポートは、ボードにデモアプリケーションを記述するために使用されます。Windows では、シリアルポートに COM1 のような名前があります。macOS では、/dev/cu で始まります。Linux では、/dev/tty で始まります。

ボードとの通信中に使用するデフォルトボーレートを変更するには、[Default baud rate (デフォルトボーレート)] を選択します。ボーレートを増やすと、ボードをフラッシュさせるのに必要な時間を短縮できます。ハードウェアに応じて、最大 921600 までデフォルトのボーレートを増やすことができます。

5. ボードの設定をセットアップしたら、保存してメニューを終了します。

ファームウェア (ブートローダーとパーティションテーブルを含む) をビルドしてフラッシュし、シリアルコンソール出力をモニタリングするには、ターミナル (macOS と Linux) または mingw32.exe (Windows) を開きます。<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make にディレクトリを変更し、次のコマンドを実行します。

```
make flash monitor
```

make flash monitor コマンドを発行したターミナルまたはコマンドプロンプトから出力が返される と、Hello World *number* および Hello World *number* ACK メッセージが MQTT クライアントページの下部に表示されます。

```
12 1350 [MQTTEcho] Echo successfully published 'Hello World 0'  
13 1357 [Echoing] Message returned with ACK: 'Hello World 0 ACK'  
14 1852 [MQTTEcho] Echo successfully published 'Hello World 1'  
15 1861 [Echoing] Message returned with ACK: 'Hello World 1 ACK'  
16 2355 [MQTTEcho] Echo successfully published 'Hello World 2'  
17 2361 [Echoing] Message returned with ACK: 'Hello World 2 ACK'  
18 2857 [MQTTEcho] Echo successfully published 'Hello World 3'  
19 2863 [Echoing] Message returned with ACK: 'Hello World 3 ACK'
```

デモの実行が終了すると、ターミナルまたはコマンドプロンプトに次のような出力が表示されます。

```
32 6380 [MQTTEcho] Echo successfully published 'Hello World 10'  
33 6386 [Echoing] Message returned with ACK: 'Hello World 10 ACK'
```

```
34 6882 [MQTTEcho] Echo successfully published 'Hello World 11'  
35 6889 [Echoing] Message returned with ACK: 'Hello World 11 ACK'  
36 7385 [MQTTEcho] MQTT echo demo finished.  
37 7385 [MQTTEcho] ----Demo finished----
```

Bluetooth Low-Energy デモを実行する

Amazon FreeRTOS の Bluetooth Low Energy サポートは、パブリックベータリリースに含まれています。BLE デモは変更されることがあります。

Amazon FreeRTOS は、[Bluetooth Low Energy \(BLE\)](#) 接続をサポートしています。BLE を備えた Amazon FreeRTOS は、[GitHub](#) からダウンロードできます。Amazon FreeRTOS BLE ライブラリは、まだパブリックベータにあるため、ボード用のコードにアクセスするにはブランチを切り替える必要があります。`feature/ble-beta` という名前のブランチを確認します。

BLE で Amazon 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 Amazon FreeRTOS BLE Mobile SDK デモアプリケーションを実行する必要があります。

Amazon FreeRTOS BLE 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、[Amazon FreeRTOS デバイス用の Mobile SDK](#) の手順に従います。
2. モバイルデバイスで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、[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション](#) の手順に従います。

ボードで MQTT over BLE デモを実行する方法については、[MQTT over BLE demo application \(MQTT over BLE デモアプリケーション\)](#) 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 Provisioning demo application \(Wi-Fi プロビジョニングデモアプリケーション\)](#)」を参照してください。

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 ドライバーガ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クラッシュ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 のウェブサイトにある [Automatically Decoding Addresses](#) 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 のウェブサイトにある「[GDBStub の 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 Amazon 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、Amazon FreeRTOS をビルドする前にクリアすることをお勧めします。
- Window 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。ビルド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
```

- デバイスが AWS IoT への接続に問題がある場合は、aws_clientcredential.h ファイルを開き、ファイル内で設定変数が正しく定義されていることを確認してください。`clientcredentialMQTT_BROKER_ENDPOINT[]` は `<1234567890123>-ats.iot.<us-east-1>.amazonaws.com` のようになっているはずです。

Espressif ESP32-DevKitC および ESP-WROVER-KIT のデバッグコード

JTAG to USB ケーブルが必要です。USB to MPSSE 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 配線色	ESP32 GPIO ピン	JTAG 通知名
茶色 (ピン 5)	IO14	TMS
黄色 (ピン 3)	IO12	TDI
黒 (ピン 10)	GND	GND
オレンジ (ピン 2)	IO13	TCK
緑 (ピン 4)	IO15	TDO

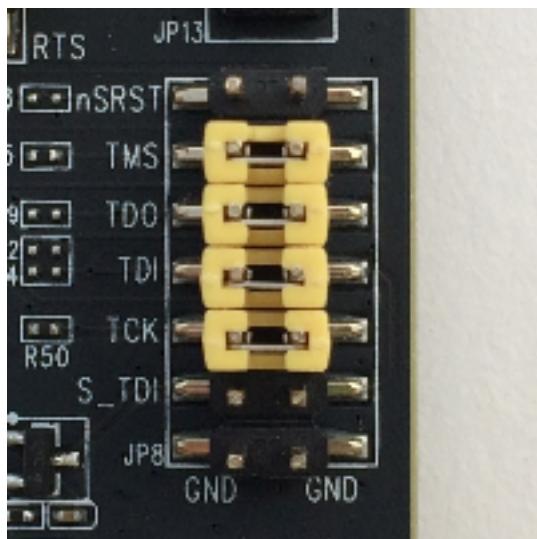
ESP-WROVER-KIT JTAG セットアップ

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 配線色	ESP32 GPIO ピン	JTAG 通知名
茶色 (ピン 5)	IO14	TMS
黄色 (ピン 3)	IO12	TDI
オレンジ (ピン 2)	IO13	TCK
緑 (ピン 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 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

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 \(p. 75\)](#) 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)] の下にある [Device Manager (デバイスマネージャ)] 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の下で、[C232HM-DDHSL-0] デバイスを右クリックし、[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。[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 `demos\espressif\esp32_devkitc_esp_wrover_kit\make \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

- 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ドロップダウンボックスから、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、[`<BASE_FOLDER>\demos\espressif\esp32_devkitc_esp_wrover_kit\make`](#) に移動して以下を実行します。

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

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、msys32 ディレクトリに移動して mingw32.exe を実行します。mingw32 ターミナルで、[`<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit\make`](#) に移動して make flash monitor を実行します。
9. 別の mingw32 ターミナルを開き、[`<BASE_FOLDER>\demos\espressif\esp32_devkitc_esp_wrover_kit\make`](#) に移動し、ボードでデモの実行が開始されるまで待機します。このときに、`xtensa-esp32-elf-gdb -x gdbinit build/aws_demos.elf` を実行します。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 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をアンロードします。

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

6. 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
```

7. 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、次のようにになります。

DEVICE:

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

8. オープン `demos/espressif/esp32_devkitc_esp_wrover_kit/make/esp32_devkitj_v1.cfg`.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。
9. ターミナルウィンドウを開き、`<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` に移動して、以下のコマンドを使用して OpenOCD を実行します。

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

10. 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をロードします。

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

11. `<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` に移動して、以下のコマンドを実行します。

```
make flash monitor
```

12. 別の新しいターミナルを開き、`<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make` に移動して、次のコマンドを実行します。

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

`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

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. ターミナルウインドウで <BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make に移動して、以下のコマンドを使用して OpenOCD を実行します。

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

6. 別のターミナルを開き、<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make に移動して次のコマンドを実行します。

```
make flash monitor
```

7. 別のターミナルを開き、<BASE_FOLDER>/demos/espressif/esp32_devkitc_esp_wrover_kit/make に移動して次のコマンドを実行します。

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

main() でこのプログラムを停止する必要があります。

Infineon XMC4800 IoT 接続キットの開始方法

Infineon XMC4800 IoT 接続キットがない場合は、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。

環境をセットアップする

Amazon 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 (たとえば、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 ドライバをインストールする

XMC4800 Relax EtherCAT ボードのオンボードデバッガー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。

シリアル接続を設定する

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 デバッガー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

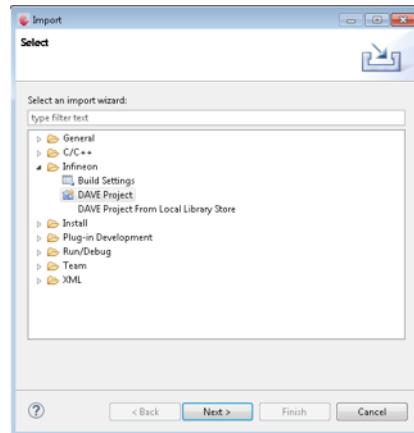
一部のシリアルケーブルは、5V シグナルレベルを使用します。XMC4800 ボードと Wi-Fi クリックモジュールには、3.3V が必要です。ボードのシグナルを 5V に変更するために、ボードの IOREF ジャンパーを使用することはしないでください。

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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

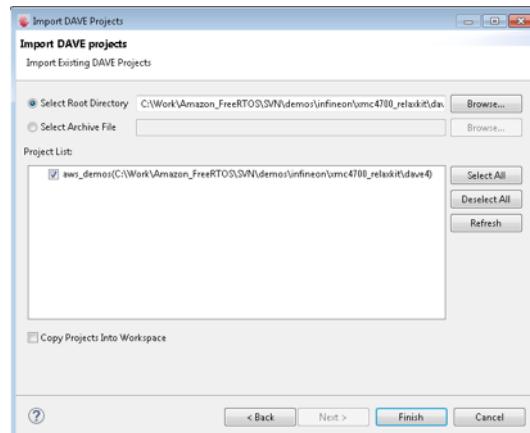
Amazon FreeRTOS デモを DAVE にインポートする

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。



- [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。

Amazon 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 `<BASE_FOLDER>/demos/infineon/xmc4800_iotkit/dave` 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。

- [Finish] 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。
- [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ビルド)] 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。

Amazon FreeRTOS デモプロジェクトを実行する

- 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。
- [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ビルダ)] を選択して、aws_demos を再ビルトし、設定変更が取得されたことを確認します。
- [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 (デバッガ方法)] 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 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、デバイスから送信された MQTT メッセージがステップ 4~5 の MQTT クライアントに表示されます。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 freertos/demos/echo
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----
```

トラブルシューティング

MediaTek MT7697Hx Development Kit の開始方法

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。

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 Amazon FreeRTOS プロジェクトを設定、ビルト、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、[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。

シリアル接続を設定する

MediaTek MT7697Hx Development Kit とのシリアル接続を確立するには、Arm Mbed Windows シリアルポートドライバをインストールする必要があります。ドライバは、[Mbed](#) からダウンロードできます。Windows シリアルドライバページのステップに従って、MediaTek MT7697Hx Development Kit のドライバをダウンロードしてインストールします。

ドライバのインストールが完了すると、Windows Device Manager 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。

Keil MDK を使用して Amazon FreeRTOS デモプロジェクトをビルトおよび実行する

Keil μVision で Amazon FreeRTOS デモプロジェクトをビルトするには

1. [スタート] メニューから Keil μVision 5 を開きます。
2. `<BASE_FOLDER>/demos/mediatek/mt7697hx-dev-kit/uvision/aws_demo.uvprojx` プロジェクトファイルを開きます。
3. メニューから、[プロジェクト]、[ビルトターゲット] の順に選択します。

コードがビルトされたら、`<BASE_FOLDER>/demos/mediatek/mt7697hx-dev-kit/uvision/out/Objects/aws_demo.axf` にデモ実行ファイルが表示されます。

Amazon 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] 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] 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。[使用] を選択して、デバッガーを [CMSIS-DAP Debugger] に設定し、[OK] を選択します。
 4. メニューから、[Flash]、[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] ボタンを選択します。

トラブルシューティング

Keil μVision での Amazon FreeRTOS プロジェクトのデバッグ

現時点では、Keil μVision で MediaTek 用の Amazon 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 Amazon 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\MediaTek\MTx\4.6.1\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] を選択します。
2. [ターゲット] タブを選択し、[Read/Write Memory Areas] 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. [デバッグ] タブ、[CMSIS-DAP Debugger] の順に選択します。
4. <BASE_FOLDER>/demos/mediatek/mt7697hx-dev-kit/common/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ビルトします。
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] ボタンを押して離し、[PROG] ボタンを離します。

7. μVision メニューから、[Debug]、[Start/Stop Debug Session] 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] ウィンドウが開きます。μVision でデモからボードに点滅後、デモが実行され、main() 関数の開始時に停止します。
8. メニューから、[デバッグ]、[停止] の順に選択し、セッションを停止します。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 ) ; }
```

9. [Call Stack + Locals] ウィンドウで、`wait_ice` の値を 0 に変更します。
10. 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。

Microchip Curiosity PIC32MZEF の開始方法

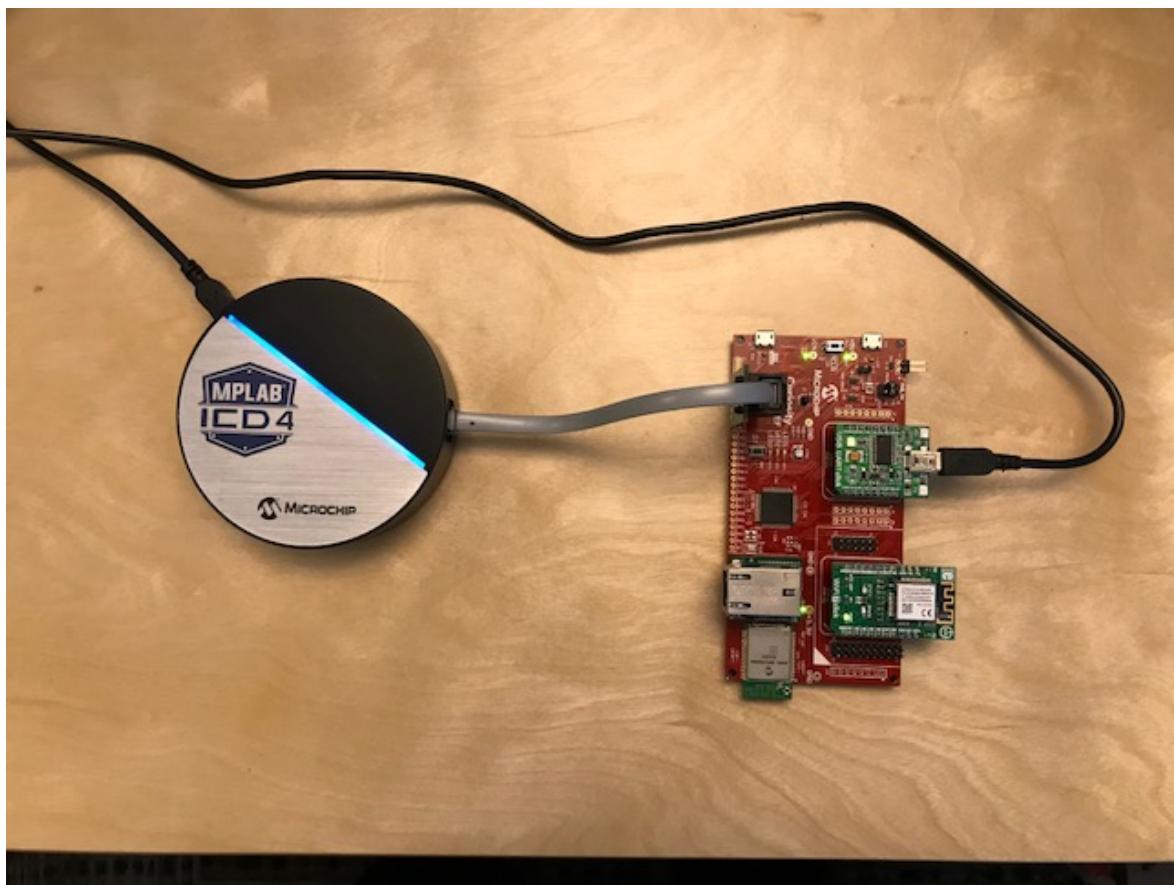
Microchip Curiosity PIC32MZEF バンドルがない場合は、AWS Partner Device Catalog にアクセスして当社のパートナーから購入してください。次のアイテムが必要です。

- [MikroElectronika USB UART Click Board](#)
- [RJ-11 to ICSP Adapter](#)
- [MPLAB ICD 4 In-Circuit Debugger](#)
- [PIC32 LAN8720 PHY daughter board](#)
- [MikroElectronika WiFi 7 Click Board](#)

Microchip Curiosity PIC32MZEF ハードウェアのセットアップ

1. [MikroElectronika USB UART Click Board](#) を Microchip Curiosity PIC32MZEF の microBUS 1 コネクタに接続します。
2. [PIC32 LAN8720 PHY daughter board](#) を Microchip Curiosity PIC32MZEF 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. [MikroElectronika WiFi 7 Click Board](#) を Microchip Curiosity PIC32MZEF の microBUS 2 コネクタに接続します。
5. Microchip Curiosity PIC32MZEF に [RJ-11 to ICSP Adapter](#) を接続します。
6. RJ-11 ケーブルを使用して、MPLAB ICD 4 インサーキットデバッガーを Microchip Curiosity PIC32MZEF に接続します。
7. USB A to USB mini-B ケーブルを使用して、ICD 4 インサーキットデバッガーをコンピュータに接続します。
8. ICSP アダプタ J2 への RJ-11 を J16 で Microchip Curiosity PIC32MZEF の ICSP ヘッダーに挿入します。
9. 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。

次の図は、Microchip Curiosity PIC32MZEF とすべての必要な周辺機器を組み立てたものです。



インサーキットデバッガーの LED は、準備が完了すると青に変わります。

環境をセットアップする

1. 最新の Java SE SDK をインストールします。
2. Python バージョン 3.x 以降をインストールします。
3. MPLAB X IDE の最新バージョンをインストールします。
 - [MPLAB X Integrated Development Environment for Windows](#)
 - [MPLAB X Integrated Development Environment for macOS](#)
 - [MPLAB X Integrated Development Environment for Linux](#)
4. MPLAB XC32 コンパイラの最新バージョン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](#)
5. MPLAB Harmony Integrated Software Framework の最新バージョンをインストールします (オプション)。
 - [MPLAB Harmony Integrated Software Framework for Windows](#)
 - [MPLAB Harmony Integrated Software Framework for macOS](#)
 - [MPLAB Harmony Integrated Software Framework for Linux](#)
6. UART ターミナルエミュレータを起動し、以下の設定で接続を開きます。

- ポーレート: 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

Amazon FreeRTOS デモプロジェクトを構築および実行する

MPLAB IDE で Amazon FreeRTOS デモアプリケーションを開く

1. MPLAB IDE の [File (ファイル)] メニューから、[Open Project (プロジェクトを開く)] を選択します。
2. <BASE_FOLDER>\demos\microchip\curiosity_pic32mzef\mplab に移動して開きます。
3. [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、次のような警告メッセージは無視してかまいません。

```
warning: Configuration "pic32mz_ef_curiosity" builds with "XC32", but indicates no
         toolchain directory.
warning: Configuration "pic32mz_ef_curiosity" refers to file "AmazonFreeRTOS/lib/
         third_party/mcu_vendor/microchip/harmony/framework/bootloader/src/bootloader.h"
         which does not exist in the disk. The make process might not build correctly.
```

Amazon FreeRTOS デモプロジェクトを実行する

1. プロジェクトを再ビルドします。
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. サンプルを初めてデバッグすると、[ICD 4 not Found (ICD 4 が見つかりません)] ダイアログボックスが表示されます。ツリービューの [ICD 4] ノードから、ICD4 シリアル番号を選択して [OK] 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。

ICD 4 は、デバイスをプログラミングしているときは半分黄色に変わり、実行中は半分緑色に変わります。[ICD4] は、IDE に表示されます。プログラミングに成功すると、以下のようになります。

```
*****
Connecting to MPLAB ICD 4...
Currently loaded versions:
Application version.....01.02.00
Boot version.....01.00.00
FPGA version.....01.00.00
Script version.....00.02.18
Script build number.....fd44437f19
```

```
Application build number.....0123456789
Connecting to MPLAB ICD 4...
Currently loaded versions:
Boot version.....01.00.00
Updating firmware application...
Connecting to MPLAB ICD 4...
Currently loaded versions:
Application version.....01.02.16
Boot version.....01.00.00
FPGA version.....01.00.00
Script version.....00.02.18
Script build number.....fd44437f19
Application build number.....0123456789
Target voltage detected
Target device PIC32MZ2048EFM100 found.
Device Id Revision = 0xA1
Serial Number:
Num0 = ec4f6d3c
Num1 = 6b845410
Erasing...
The following memory area(s) will be programmed:
program memory: start address = 0x1d000000, end address = 0x1d07bfff
program memory: start address = 0x1d1fc000, end address = 0x1d1fffff
configuration memory
boot config memory
Programming/Verify complete
Running
```

Note

デバッグには、USB ポートの代わりに MPLAB インサーキットデバッガーの使用をお勧めします。ICD 4 を使用すると、コードをすばやく進めることができ、デバッガーを再起動せずにブレークポイントを追加することができます。

トラブルシューティング

Nordic nRF52840-DK の開始方法

Amazon FreeRTOS の Nordic nRF52840-DK サポートは、パブリックベータリリースに含まれています。BLE デモは変更されることがあります。

開始する前に、[Amazon FreeRTOS BLE 用の AWS IoT および Amazon Cognito のセットアップ \(p. 183\)](#) することが必要です。

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。

Amazon FreeRTOS BLE 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 デバイスが必要です。

Nordic ハードウェアのセットアップ[®]

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。

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

Amazon 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。

Amazon FreeRTOS BLE Mobile SDK デモアプリケーションをセットアップする

BLE で Amazon FreeRTOS デモプロジェクトを実行するには、お使いのモバイルデバイスで Amazon FreeRTOS BLE Mobile SDK デモアプリケーションを実行する必要があります。

Amazon FreeRTOS BLE 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、「[Amazon FreeRTOS Bluetooth デバイス用の Mobile SDK \(p. 157\)](#)」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、「[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#)」の手順に従います。

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、[ターゲット]、[Connect J-Link] の順に選択します。
3. [ターミナルエミュレータをインストールする \(p. 66\)](#) に示されているとおりに、トップメニューから、[ツール]、[ターミナルエミュレータ]、[プロパティ] の順に選択し、プロパティを設定します。
4. トップメニューから、[ツール]、[ターミナルエミュレータ]、[Connect **port** (115200,N,8,1)] の順に選択します。

Note

また、任意のターミナルツール（例：PuTTY、Tera Term、GNU Screen）でシリアル接続を確立することもできます。[ターミナルエミュレータをインストールする（p. 66）](#) 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。

Amazon 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、Amazon FreeRTOS をダウンロードすることができます。

Amazon FreeRTOS のダウンロード

Nordic nRF52840-DK 用の Amazon FreeRTOS をダウンロードするには、[Amazon 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。Amazon FreeRTOS BLE ライブラリは、まだパブリックベータにあるため、Nordic nRF52840-DK ボード用のコードにアクセスするにはブランチを切り替える必要があります。`feature/ble-beta` という名前のブランチを確認します。

Note

Microsoft Windows でのファイルパスの最大長は 260 文字です。Amazon FreeRTOS ダウンロードでの最長パスは 122 文字です。Amazon FreeRTOS プロジェクトのファイルに対応するため、AmazonFreeRTOS ディレクトリへのパスが 98 文字未満であることを確認してください。たとえば、`C:\Users\Username\Dev\AmazonFreeRTOS` の使用は可能ですが、`C:\Users\Username\Documents\Development\Projects\AmazonFreeRTOS` というパスではビルドに失敗します。

このチュートリアルでは、ディレクトリへのパスを `AmazonFreeRTOS` 表記します。

プロジェクトの設定

デモを実行するには、プロジェクトを、AWS IoT と連携するよう設定する必要があります。AWS IoT と連携するようプロジェクトを設定するには、デバイスを AWS IoT のモノとして登録する必要があります。[Amazon FreeRTOS BLE 用の AWS IoT および Amazon Cognito のセットアップ（p. 183）](#)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](#)」を参照します。
2. ナビゲーションペインで [Settings] を選択します。

AWS IoT エンドポイントは、[Endpoint (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 モノ名を書き留めます。
4. AWS IoT エンドポイントと AWS IoT モノ名を手元に用意し、IDE の `<BASE_FOLDER>\demos\common\include\aws_clientcredential.h` を開いて次の #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. BLE GATT デモが有効になっていることを確認します。`<BASE_FOLDER>\demos\nordic\nrf52840-dk\common\config_files\aws_ble_config.h` に移動し、`bleconfigENABLE_GATT_DEMO` が 1 に設定されていることを確認します。

2. <BASE_FOLDER>\demos\common\demo_runner\aws_demo_runner.c を開き、デモの宣言の中で `extern void vStartMQTTBLEEchoDemo(void);` のコメントを解除します。DEMO_RUNNER_RunDemos 定義で、`vStartMQTTBLEEchoDemo()`; のコメントを解除します。

Amazon FreeRTOS デモプロジェクトを構築および実行する

Amazon 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ビルドして実行する準備が整いました。

Segger Embedded Studio から Amazon FreeRTOS BLE デモをビルドして実行するには

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 <BASE_FOLDER>\demos\nordic\nrf52840-dk\ses\aws_demos_ble.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 [ツール] を選択後、[ターミナルエミュレータ]、[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] の aws_ble_demos デモプロジェクトを右クリックし、[ビルド] 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。[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. [デバッグ]、[Go] の順に選択します。
5. モバイル MQTT プロキシとして Amazon FreeRTOS BLE Mobile SDK デモアプリケーションを使用してデモを完了するには、「[MQTT over BLE Demo Application](#)」の指示に従ってください。

トラブルシューティング

NXP LPC54018 IoT モジュールの開始方法

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。

環境をセットアップする

Amazon 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. [[Software for NXP Kit \(NXP キット用ソフトウェア\)](#)] を参照し、[Download Software (ソフトウェアのダウンロード)] を選択します。

Note

- IAR Embedded Workbench for ARM には、Microsoft Windows が必要です。
2. インストーラを解凍し、実行します。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。

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。
2. [[MCUXpresso SDK](#)] を参照して、[Build your SDK (SDK をビルド)] を選択します。
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 MCUXpresso SDK (MCUXpresso SDK のビルド)] 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] 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。

JTAG デバッガーの接続

NXP LPC54018 board 上で実行されてい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。Amazon FreeRTOS は Segger J-Link プローブを使用してテストされています。サポートされているデバッガーの詳細については、「[NXP LPC54018 ユーザーガイド](#)」を参照してください。

Note

Segger J-Link デバッガーを使用している場合、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するためのコンバーターケーブルが必要です。

Amazon FreeRTOS デモプロジェクトを構築および実行する

Amazon FreeRTOS デモを IDE にインポートする

Amazon FreeRTOS サンプルコードを 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)] メニューから、[Open Workspace (WorkSpace を開く)] を選択します。
2. [search-directory (検索ディレクトリ)] のテキストボックスに「**<BASE_FOLDER>\demos\ntp\lpc54018_iot_module\iar**」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 (すべて再ビルド)] を選択します。

Amazon 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「`<BASE_FOLDER>\demos\nxp\lpc54018_iot_module\mcuxpresso`」と入力し、[Finish (完了)] を選択します。
3. [Project (プロジェクト)] メニューから [Build All (すべてビルド)] を選択します。

Amazon FreeRTOS デモプロジェクトを実行する

1. NXP IoT モジュールの USB ポートをホストコンピュータに接続し、ターミナルプログラムを開いて USBシリアルデバイスとして識別されたポートに接続します。
2. IDE で、[Project (プロジェクト)] メニューから [Build (ビルド)] を選択します。
3. 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 IoT モジュールと Segger J-Link デバッガーを接続します。
4. IAR Embedded Workbench を使用している場合。
 - a. [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。
 - b. [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。
 - c. 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。

5. MCUXpresso を使用している場合。
 - a. デバッグを初めて使用する場合は、[Debug (デバッグ)] ツールバーから `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。
 - b. 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。)

6. 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

Renesas Starter Kit+ for RX65N-2MB の開始方法

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。

Renesas ハードウェアのセットアップ

RSK+ for RX65N-2MB をセットアップするには

1. RSK+ for RX65N-2MB の USB-UART ポートにコンピュータを接続します。
2. 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. 中央の正の +5V 電源アダプタを RSK+ for RX65N-2MB の PWR コネクタに接続します。
4. 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のインストールが開始されます。

ドライバ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。

環境をセットアップする

RSK+ for RX65N-2MB の Amazon FreeRTOS 設定をセットアップするには、Renesas e²studio IDE および CC-RX コンパイラを使用します。

Note

Renesas e²studio 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、ユーザー名とパスワードを入力し、[Login (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。

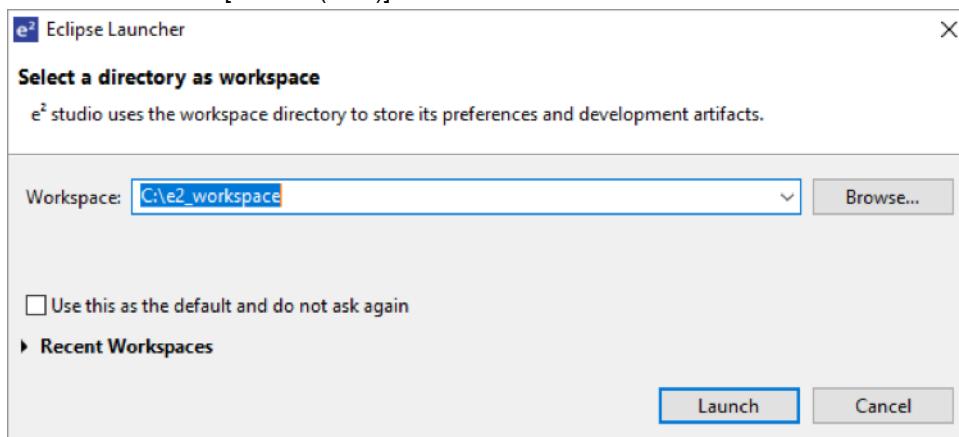
Amazon FreeRTOS サンプルをビルドおよび実行する

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 Amazon FreeRTOS デモをビルドする

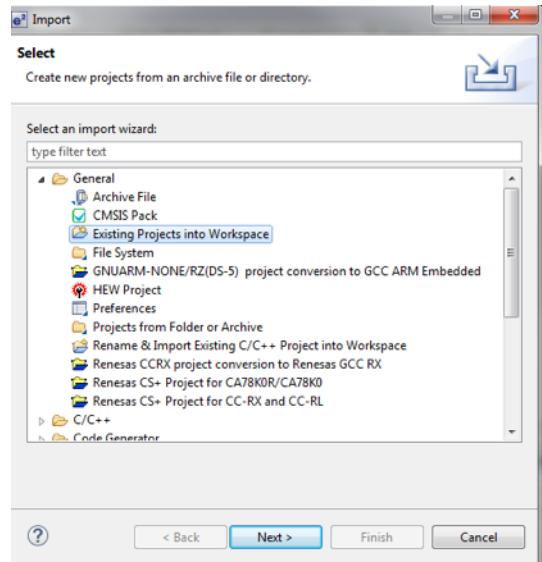
e²studio でデモをインポートしてビルドするには

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、作業するフォルダーを参照し、[Launch (起動)] を選択します。

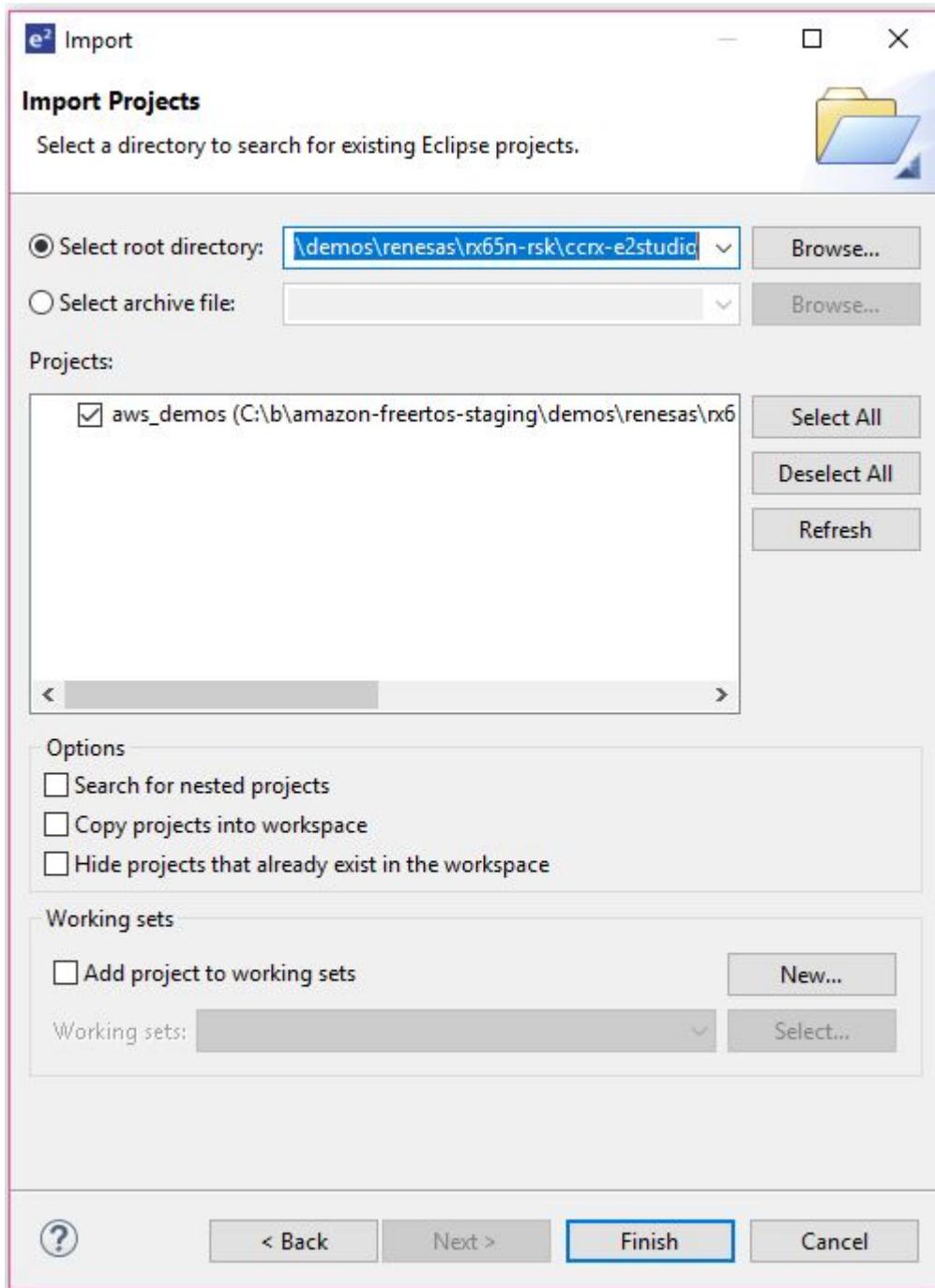


3. e²studio を始めて開くと、[Toolchain Registry (ツールチェーン登録)] ウィンドウが開きます。[Renesas Toolchains (Renesas ツールチェーン)] を選択し、CC-RX v3.00.00 が選択されていることを確認します。[Register (登録)]、[OK] の順に選択します。
4. e²studio を始めて開いている場合、[Code Generator Registration (コードジェネレーター登録)] ウィンドウが表示されます。[OK] を選択します。
5. [Code Generator COM component register (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] を選択します。
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] 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、<BASE_FOLDER>\demos\renesas\rx65n-rsk\ccrx-e2studio というディレクトリを見つけて、[Finish (完了)] を選択します。



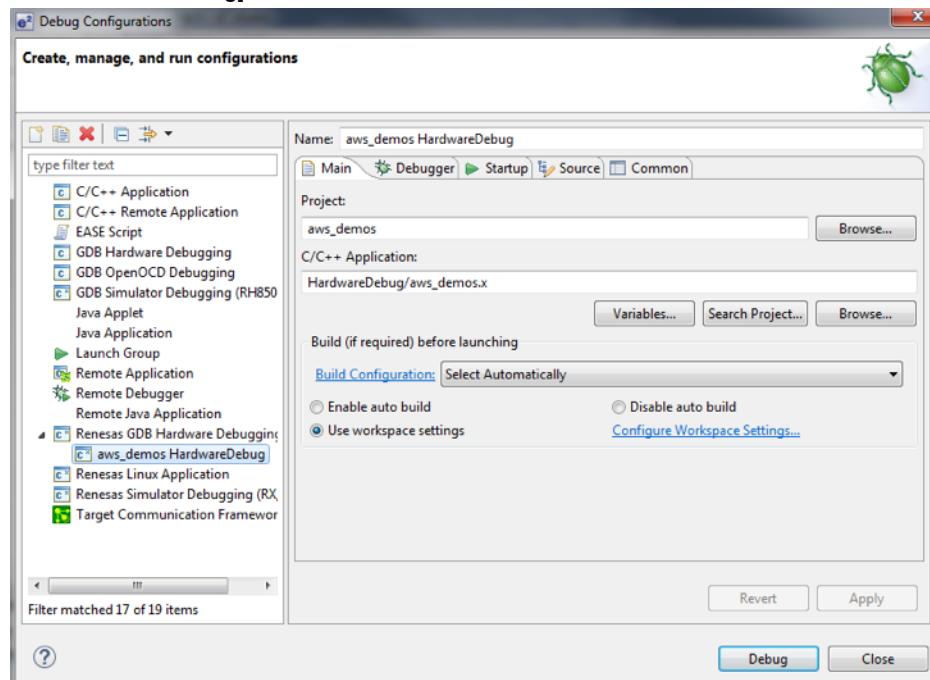
12. [Project (プロジェクト)] メニューから、[Project (プロジェクト)]、[Build All (すべてビルド)] を選択します。

ビルドコンソールに、ライセンスマネージャー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。ライセンスマネージャーをインストールするには、[ライセンスマネージャー](#)のダウンロードページを参照してください。

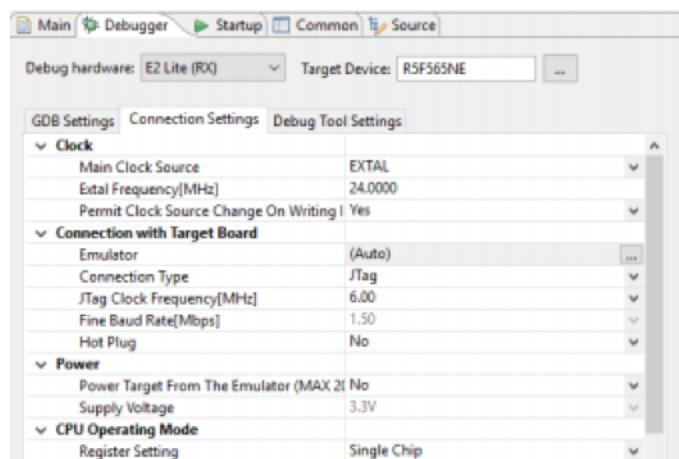
Amazon FreeRTOS プロジェクトを実行する

e²studio 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. e²studio 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] 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 によって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

STMicroelectronics STM32L4 ディスカバリキット IoT ノード用の開始方法

STMicroelectronics STM32L4 ディスカバリキット IoT ノードがない場合は、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。

最新の Wi-Fi フ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、BLE、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。

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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

Amazon 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、「<BASE_FOLDER>\demos\st\stm321475_discovery\ac6」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 (すべてビルド)] を選択します。プロジェクトがエラー や警告なしでコンパイルされているかどうかを確認します。

Amazon 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。
2. プロジェクトを再ビルドします。
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 (デバッガ方法)] 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] を選択し、[Debug] 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。

Bluetooth Low-Energy デモを実行する

Amazon FreeRTOS の Bluetooth Low Energy サポートは、パブリックベータリリースに含まれています。BLE デモは変更されることがあります。

Note

BLE デモを実行するには、STM32L475 ディスカバリキット用の SPBTLE-1S BLE モジュールが必要です。

Amazon FreeRTOS は、[Bluetooth Low Energy \(BLE\)](#) 接続をサポートしています。BLE を備えた Amazon FreeRTOS は、[GitHub](#) からダウンロードできます。Amazon FreeRTOS BLE ライブラリは、まだパブリックベータにあるため、ボード用のコードにアクセスするにはブランチを切り替える必要があります。feature/ble-beta という名前のブランチを確認します。

BLE で Amazon 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 Amazon FreeRTOS BLE Mobile SDK デモアプリケーションを実行する必要があります。

Amazon FreeRTOS BLE 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、[Amazon FreeRTOS デバイス用の Mobile SDK](#) の手順に従います。

- モバイルデバイスで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、Amazon FreeRTOS BLE Mobile SDK デモアプリケーションの手順に従います。

ボードで MQTT over BLE デモを実行する方法については、MQTT over BLE demo application (MQTT over BLE デモアプリケーション) 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: xxxxxxxxxxxxxxxx
[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、BLE、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。

Texas Instruments CC3220SF-LAUNCHXL の開始方法

Texas Instruments (TI) CC3220SF-LAUNCHXL 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パートナーから購入してください。

環境をセットアップする

以下のステップに従って、Amazon 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。

Amazon FreeRTOS は、TI CC3220SF-LAUNCHXL Development Kit 用の 2 つの IDE (Code Composer Studio と IAR Embedded Workbench) 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

- TI Code Composer Studio を参照します。
- 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 (バージョン 7.3.0) をダウンロードします。

Note

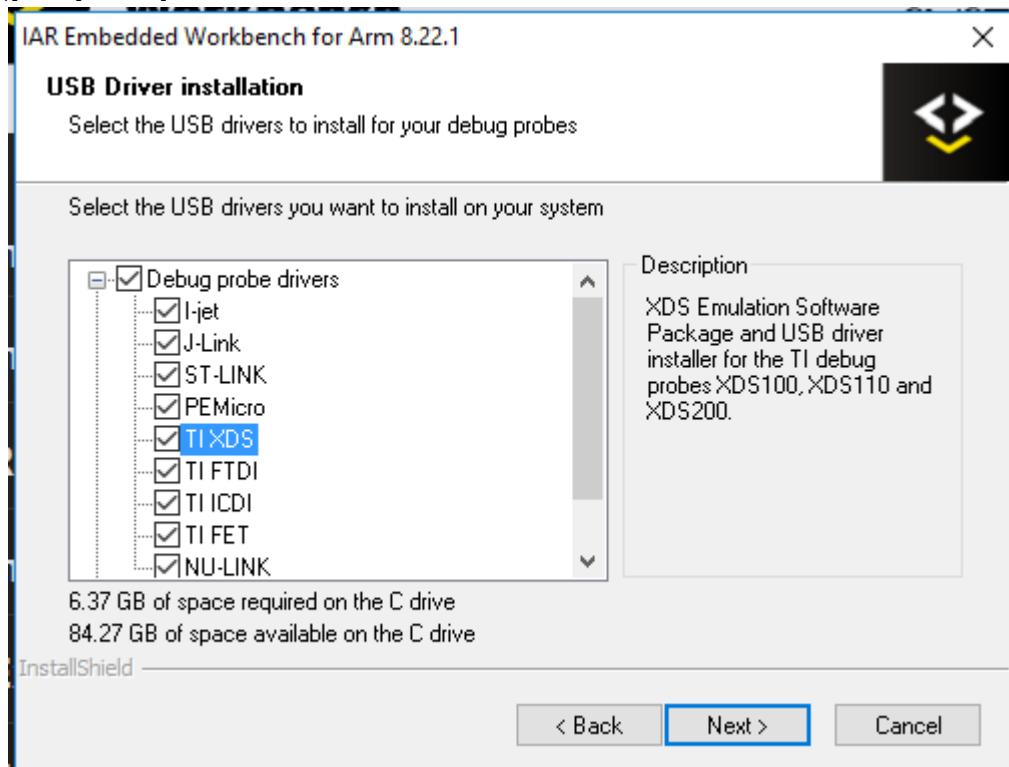
現在、Amazon FreeRTOS デモコードは TI Code Composer Studio のバージョン 7.3.0 とのみ互換性があります。Code Composer Studio の新しいバージョンではビルトエラーが発生する可能性があります。

- 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。
- [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。
- 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](#) を参照します。
2. 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。



3. 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、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、[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)] の下のボードの一覧から [CC3220SF-LAUNCHXL] を選択し、次に [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 (デバイスマード)] 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。

- [Amazon FreeRTOS デモを設定する \(p. 64\)](#) に示されている Amazon 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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

TI Code Composer で Amazon FreeRTOS デモプロジェクトをビルドおよび実行する

Amazon 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、「`<BASE_FOLDER>\demos\ti\cc3220_launchpad\ccs`」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 Amazon 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、「[CC3220 SimpleLink 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。

Note

Amazon FreeRTOS デモには 8 つのデバッグメッセージが設定されています。

IAR Embedded Workbench で Amazon FreeRTOS デモプロジェクトをビルドして実行する

Amazon 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. `<BASE_FOLDER>\demos\ti\cc3220_launchpad\iar` 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 Amazon 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、「[CC3220 SimpleLink 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. プロジェクトを再ビルトします。
プロジェクトを再ビルトするには、[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。

Note

Amazon FreeRTOS デモには 8 つのデバッグメッセージが設定されています。

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生じる場合があります。

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], [Close] 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。

Windows Device Simulator の開始方法

Amazon FreeRTOS は、指定したプラットフォーム用の Amazon 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ました。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。

環境をセットアップする

1. WinPCap の最新バージョンをインストールします。
2. [Desktop development with C++ (C++ を使用したデスクトップ開発)] ワークフローを使用して Microsoft Visual Studio Community 2017 をインストールし、Windows 8.1 SDK をインストールします。
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。

ネットワークインターフェイスを設定する

1. Visual Studio で、プロジェクトを実行します。プログラムは、ネットワークインターフェイスを列挙します。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 Attempting to open interface number 1.
```

[Cannot find or open the PDB file (PDB ファイルが見つからないか、または開けません)] という出力がデバッガに表示される場合があります。これらのメッセージは無視できます。

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。

2. <BASE_FOLDER>\demos\pc\windows\common\config_files\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。

Amazon FreeRTOS デモプロジェクトを構築および実行する

Visual Studio に Amazon FreeRTOS デモをロードする

1. Visual Studio の [File (ファイル)] メニューから、[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、<**BASE_FOLDER**>\demos\pc\windows\visual_studio\aws_demos.sln に移動してから [Open (開く)] を選択します。
2. [Build (ビルド)] メニューから [Build Solution (ソリューションのビルド)] を選択し、ソリューションがエラーや警告なしでビルドされることを確認します。

Amazon FreeRTOS デモを実行する

1. Visual Studio プロジェクトを再ビルドし、ヘッダーファイルで加えられた変更を取得します。
2. Visual Studio の [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。

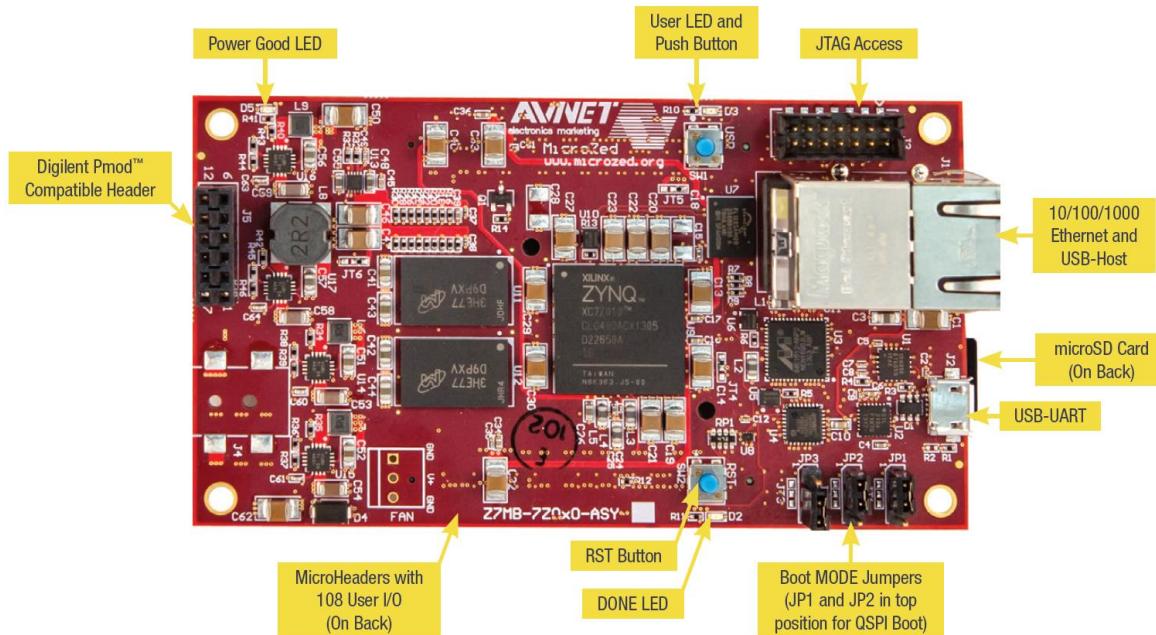
トラブルシューティング

Xilinx Avnet MicroZed Industrial IoT キットの開始方法

Xilinx Avnet MicroZed Industrial IoT キットがない場合は、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。

MicroZed ハードウェアのセットアップ[®]

次の図は、MicroZed ハードウェアをセットアップ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。

環境をセットアップする

MicroZed キットの Amazon 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](#)から [WebInstall クライアント](#) 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。

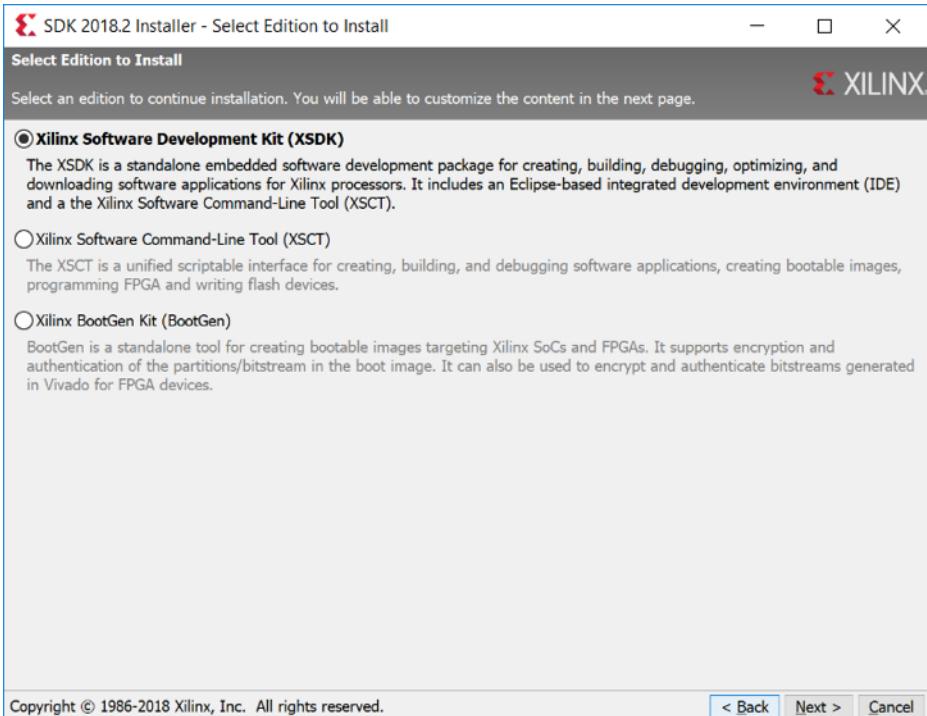
Xilinx のアカウントを持っている場合は、ユーザー名とパスワードを入力し、[Sign in (サインイン)] を選択します。

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。

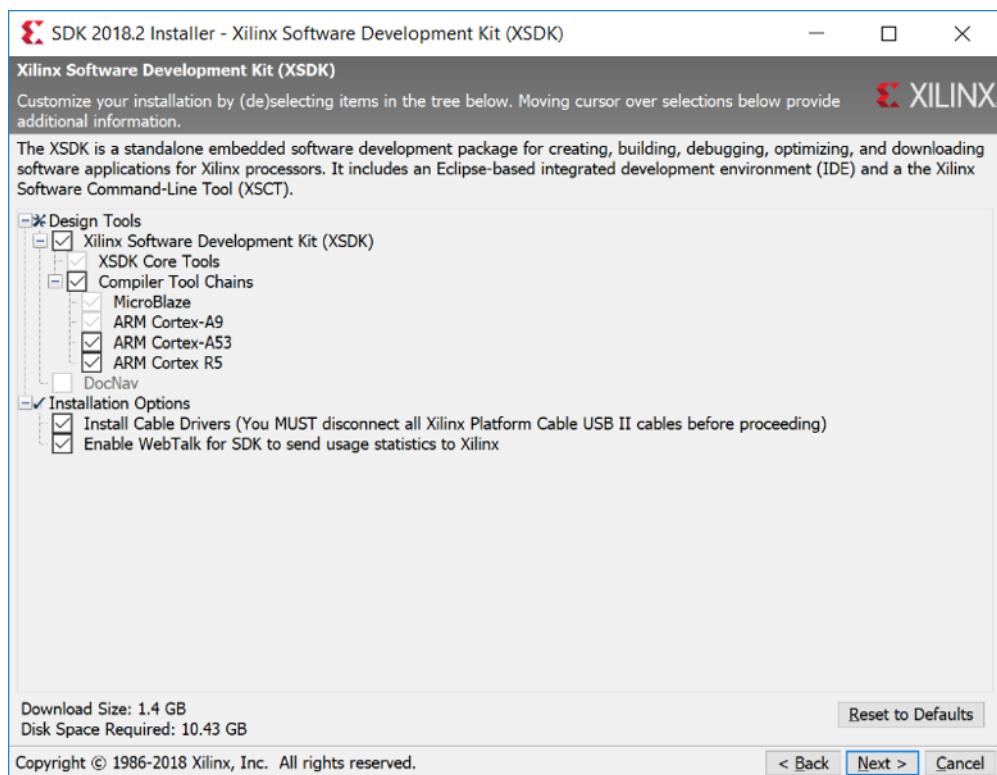
4. [Name and Address Verification (名前と住所の検証)] ページで、情報を入力して [Next (次へ)] を選択します。ダウンロードを開始する準備ができているはずです。
5. `Xilinx_SDK_version_os` ファイルを保存します。

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 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 [インストールオプション] で、[Install Cable Drivers (ケーブルドライバのインストール)] を選択し、[Next (次へ)] を選択します。



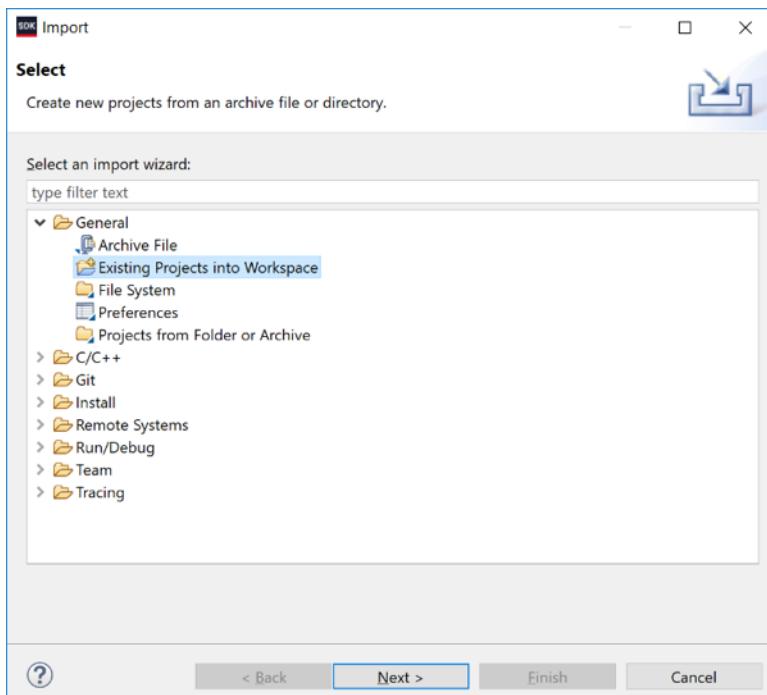
コンピュータが MicroZed の USB-UART 接続を検出しない場合は、CP210x USB-to-UART Bridge VCP ドライバを手動でインストールしてください。手順については、「[Silicon Labs CP210x USB-to-UART インストールガイド](#)」を参照してください。

XSDK の詳細については、Xilinx のウェブサイトの「[Xilinx SDK の開始方法](#)」を参照してください。

Amazon FreeRTOS デモプロジェクトを構築および実行する

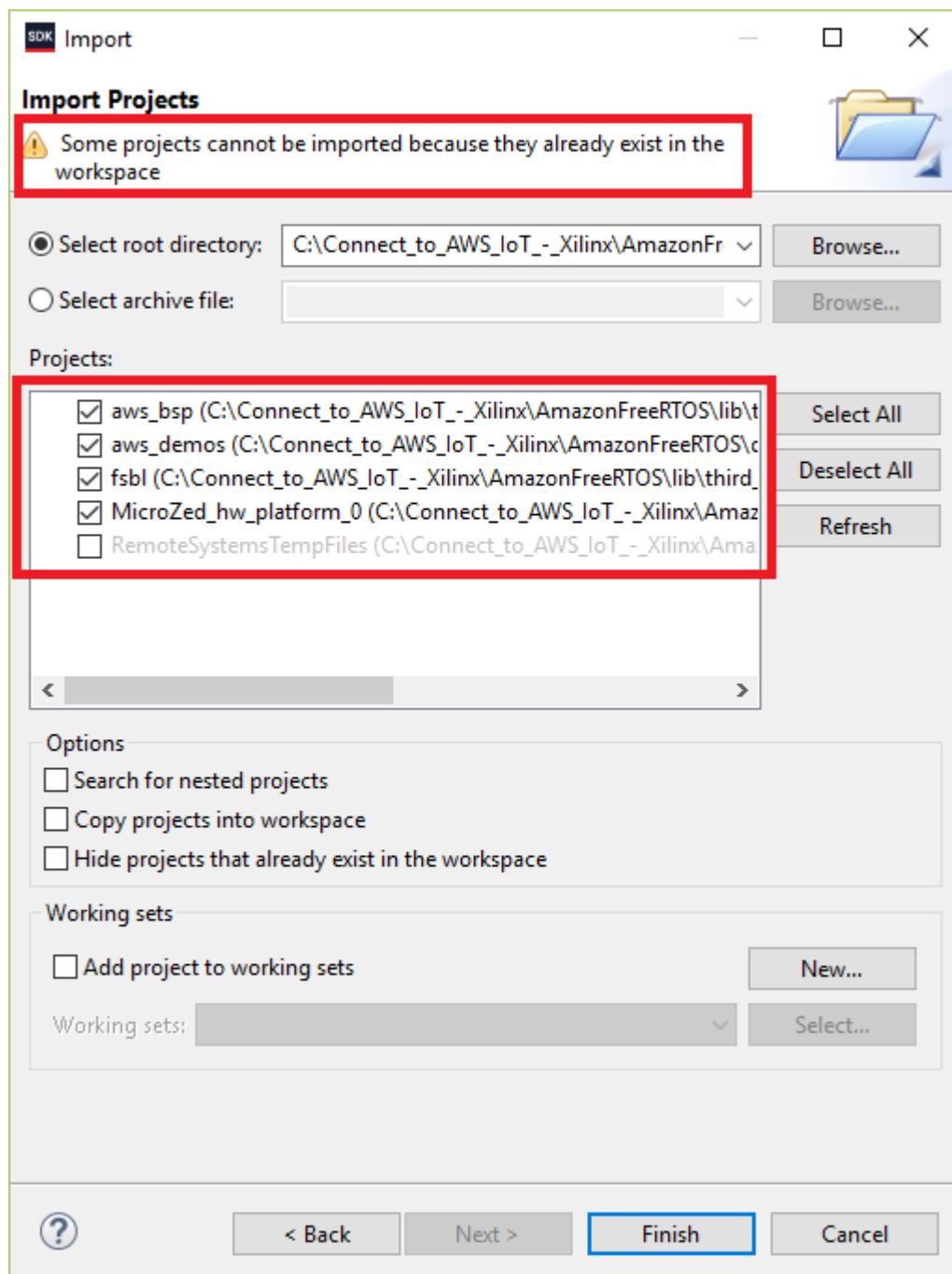
XSDK IDE で Amazon FreeRTOS デモを開く

- ワークスペースディレクトリを `<BASE_FOLDER>\demos\xilinx\microzed\xsdk` に設定して XSDK IDE を起動します。
- 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 (自動的にビルドする)] をクリアします。
- 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。
- [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。

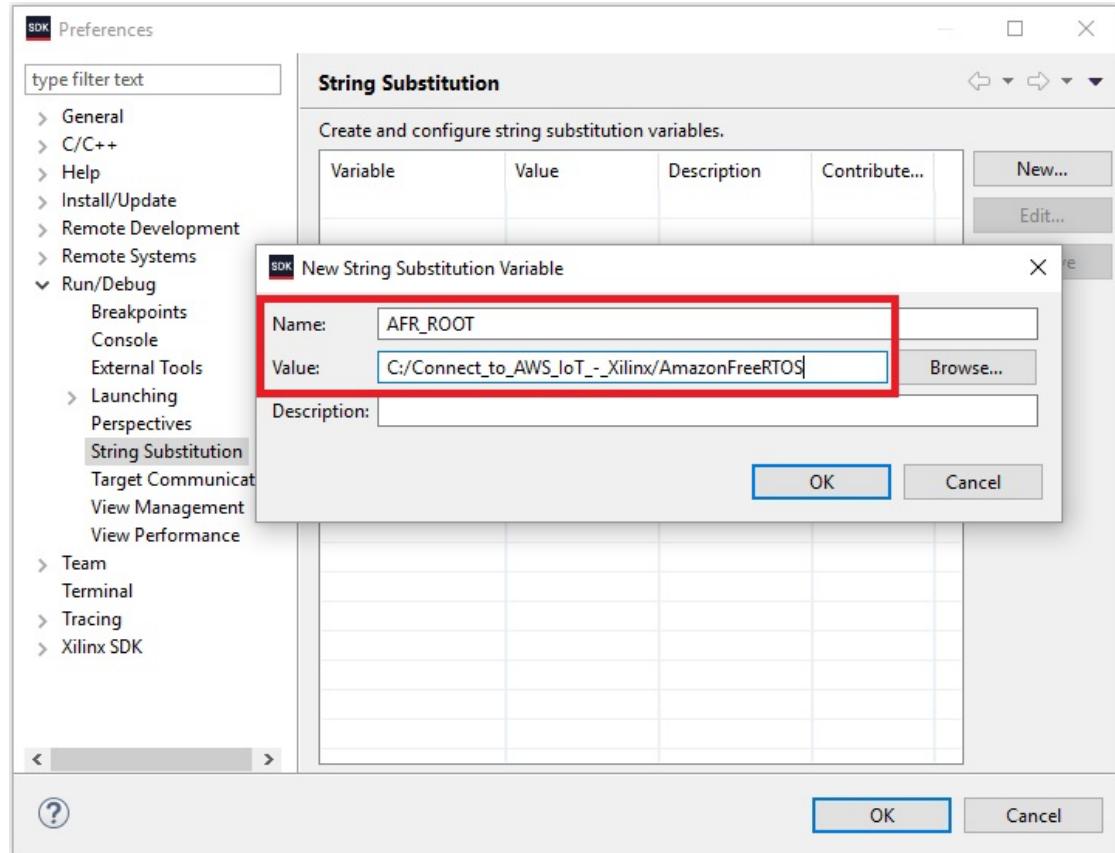


Note

[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ベ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、無視してください。

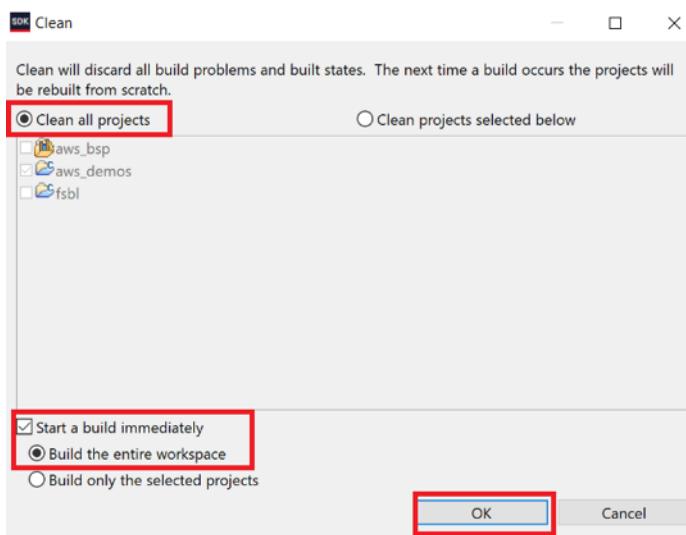
6. 選択されたすべてのプロジェクトで、[Finish (終了)] を選択します。XSDK IDE は、aws_demos プロジェクトが MicroZed ボードでビルドおよび実行するために必要なすべてのプロジェクトを開きます。
7. メニューから、[Window (ウィンドウ)] を選択し、[Preferences (プリファレンス)] を選択します。

8. 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。
9. [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、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。



Amazon FreeRTOS デモプロジェクトをビルドする

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ビルドするには、[Project (プロジェクト)] を選択し、[Build All (すべてビルド)] を選択します。
個々のプロジェクトをビルドするには、ビルド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 (プロジェクトをビルドする)] を選択します。

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。

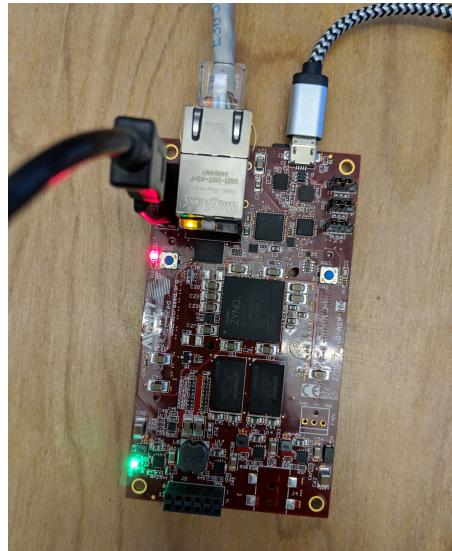


2. MicroSD カードを USB-UART ポートのすぐ下にある MicroSD カードスロットに挿入します。

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。



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、[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。

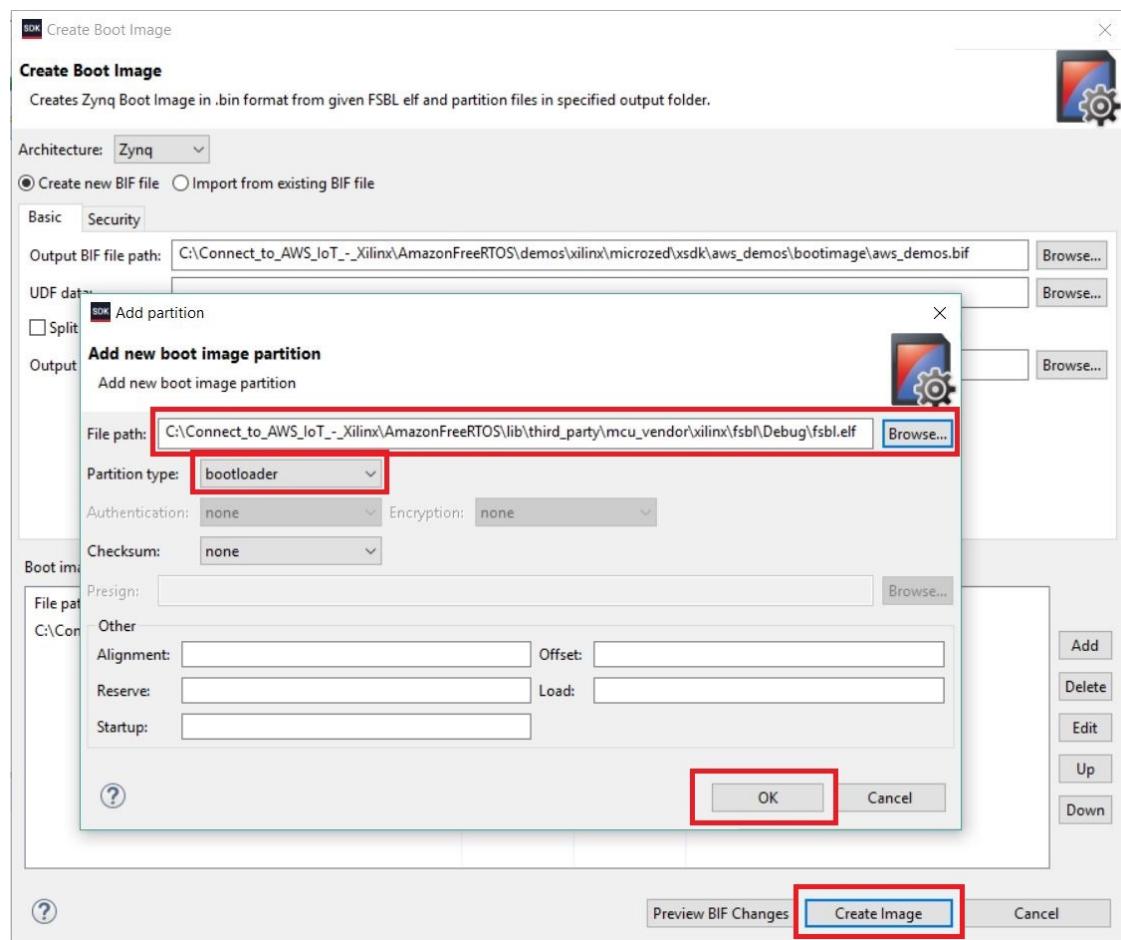
Note

初めてアプリケーションを実行する際に、新しい証明書のキーペアが生成されます。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 ファイルでの 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、Amazon FreeRTOS デモプロジェクトを実行することができます。手順については、「[Amazon FreeRTOS デモプロジェクトから起動イメージを生成する \(p. 114\)](#)」および「[Amazon FreeRTOS デモプロジェクトを実行する \(p. 115\)](#)」を参照してください。

Amazon 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、<BASE_FOLDER>\demos\xilinx\microzed\xsdk\aws_demos\bootimage\aws_demos.bif にある aws_demos.bif を選択します。
4. [Add] 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、<BASE_FOLDER>\lib\third_party\mcu_vendor\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、demos\xilinx\microzed\xsdk\aws_demos\bootimage\BOOT.bin に BOOT.bin ファイルを生成します。

Amazon FreeRTOS デモプロジェクトを実行する

Amazon 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。

Amazon FreeRTOS デモプロジェクトを実行するため MicroZed ボードを設定するときは、[MicroZed ハードウェアのセットアップ \(p. 106\)](#) 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 Amazon 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。

SD Card



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 Amazon FreeRTOS デモプロジェクトを起動する

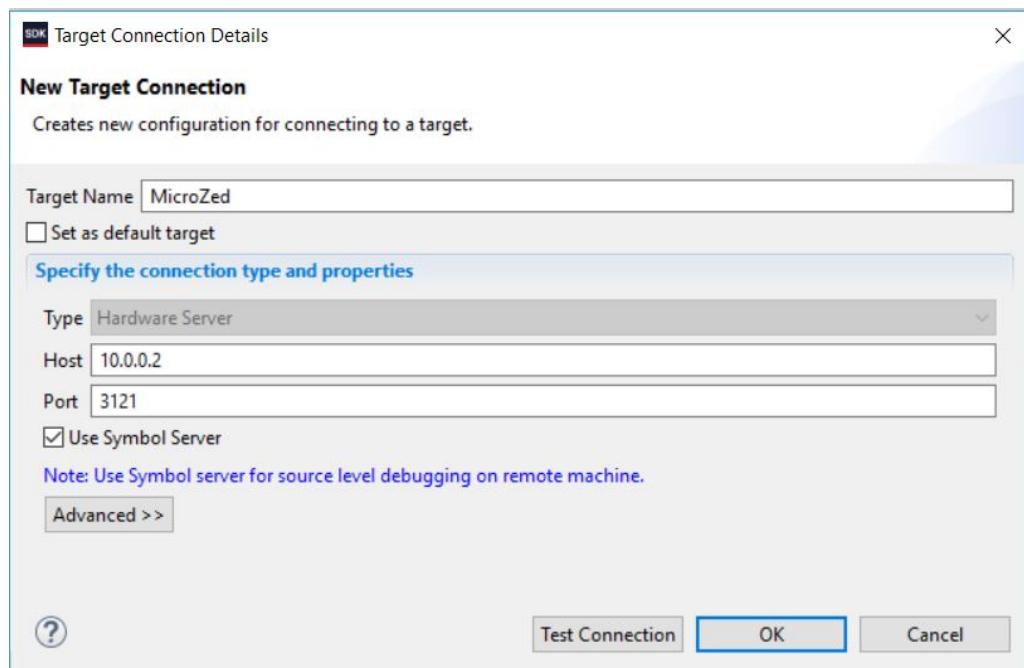
1. MicroZed ボードのブートモードジャンパーを JTAG ブートモードに設定します。



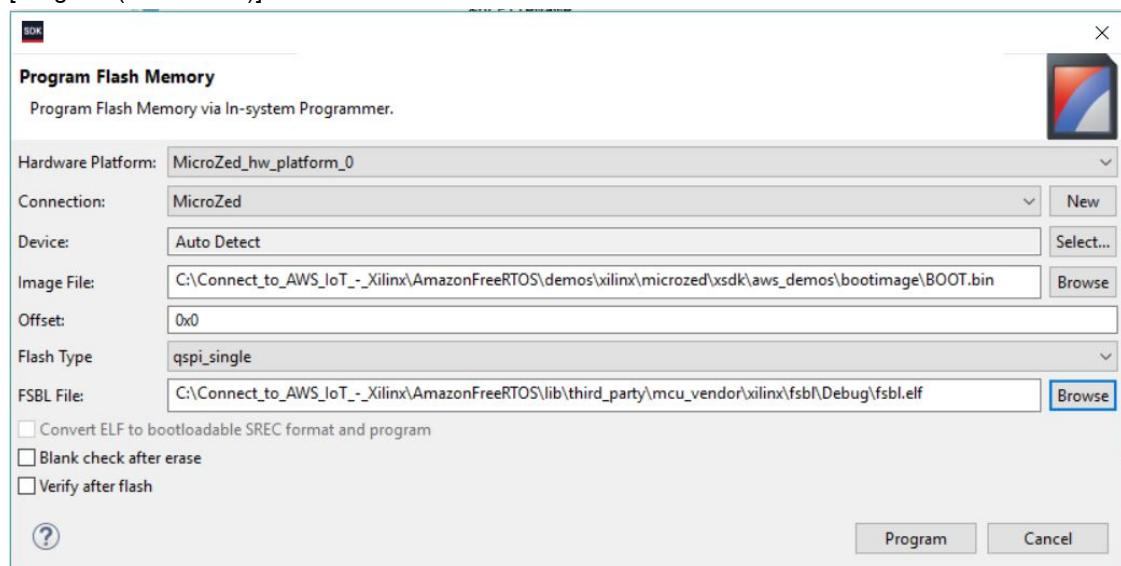
2. コンピュータが USB-UART および JTAG アクセスポートに接続されていることを確認します。緑色の Power Good LED ライトが点灯しているはずです。
3. XSDK IDE で、メニューから、[Xilinx] を選択し、[Program Flash (フラッシュのプログラム)] を選択します。
4. [Program Flash Memory (フラッシュメモリのプログラム)] で、ハードウェアプラットフォームが自動的に入力されているはずです。[Connection (接続)] で、MicroZed ハードウェアサーバーを選択して、ホストコンピュータとボードを接続します。

Note

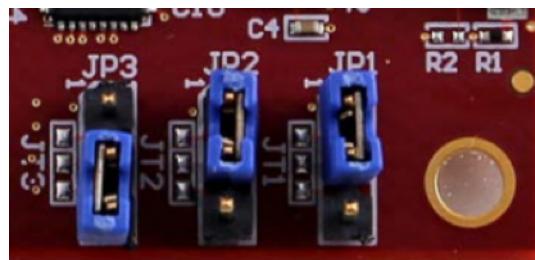
Xilinx Smart Lync JTAG ケーブルを使用している場合は、XSDK IDE にハードウェアサーバーを作成する必要があります。[New (新規)] を選択し、サーバーを定義します。



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

誤ったパスに関するビルトエラーが発生した場合は、[Amazon FreeRTOS デモプロジェクトをビルドする \(p. 112\)](#) 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トを試みます。

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。

Amazon FreeRTOS ライブラリ

Amazon FreeRTOS ライブラリは、FreeRTOS カーネルとその内部ライブラリに追加の機能を提供します。組み込みアプリケーションでは、ネットワークとセキュリティのために Amazon FreeRTOS ライブラリを使用できます。Amazon FreeRTOS ライブラリはまた、アプリケーションが AWS IoT サービスを操作できるようにします。

`lib` ディレクトリには、Amazon FreeRTOS ライブラリのソースコードが含まれています。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。これらのライブラリのいずれかを変更する必要がある場合は、`lib/include` ディレクトリで定義されたライブラリインターフェイスに準拠していることを確認してください。

Amazon FreeRTOS 移植ライブラリ

以下の移植ライブラリは、Amazon FreeRTOS のコンソールにダウンロード可能な Amazon 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。

Amazon FreeRTOS 移植ライブラリ

Bluetooth
FreeRTOS
Bluetooth (Low Energy)
Energy (BLE)
UART
USB
シリアルを使用して、マイクロコントローラーは

API
明
ア
ラ
ス
ゲ
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、
「Amazon
FreeRTOS
Bluetooth
Low
Energy
Library
(ペ
タ) (p. 148)」
を
参
照
し
て
く
だ
さ
い。

翻訳
朝
ブ
ラ
ジ
ス

Note

Amazon
FreeRTOS

BLE

ライ

ブ

ラ

リ

は

パ

ブ

リ

ク

ベ

タ

に

あ

り

ま

す。

API
明
ア
ラ
リス

無線
Amazon
FreeRTOS

AWS

IoT

OTA

無線

通

信ア

工一

ジェ

ス

ト

ラ

イ

ブ

ラ

リ

は、Amazon
FreeRTOS

デ

バイス

を

AWS

IoT

OTA

工一

ジェ

ント

に接続

します。

詳細については、

「Amazon
FreeRTOS

無線

通

図
明
ブア
ラ
ス
信
(OTA)
エー
ジエ
ント
ライ
ブ
ラ
リ (p. 164)」
を
参
照
して
く
だ
さ
い。

翻訳
朝
ア
ラ
ス

FreeRTOS

+POSIX

ア
リ
ブ
ラ
リ

を使用

して、

POSIX

準拠

のア

プ
リ
ケ
ー

シ
ョ
ン

を

Amazon

FreeRTOS

エ
コ
シ
ス
テ
ム

に

移
植

す
る

こ
と
が

で
き
ま
す。

詳
細
に
つ
い
て
は、

API
明
ア
ラ
ス

「FreeRTOS
+POSIX」
を
参
照
し
て
く
だ
さ
い。

Secure
Sockets

API
カ
ケツ
区
は、

「Amazon
FreeRTOS

セ
キュ
ア
ソ
ケッ

ト
ラ
イ
ブ
ラ

リ (p. 168)」

を
参
照
し
て
く
だ
さ
い。

翻訳
朝
ア
ラ
ス

FreeRTOS

+TCP

は、FreeRTOS

の

ア

ク

ラ

ス

ル

な

オ

ー

プ

ン

ソ

ス

で

ス

レ

ッ

ド

セ

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。

概要
初期
ファ
ルマ
ス

Amazon
FreeRTOS

API

日

ア

イ

ブ

タ

リ

を

使

用

す

る

と、

マイ

ク

ロ

コ

ン

ト

ー

ラ

ー

の

低

い

レ

ベ

ル

の

ワ

イ

ヤ

レ

ス

の

ス

タ

ク

と

の

イ

ン

タ

エ

フ

イ

ス

が

可

API
明
ア
ラ
ス
能
に
な
り
ま
す。
詳
細
に
つ
い
て
は、
「Amazon
FreeRTOS
Wi-
Fi
ライ
ブ
ラ
リ (p. 173)」
を
参
照
し
て
く
だ
さ
い。

概要
明文
ブロック
暗号化
リスト

PKCS
Amazon
FreeRTOS

PKCS

#11

ライ

ブ

ラ

リ

バ

リ

は

、

プロ

ビ

ジ

ニ

ン

グ

と

TLS

ク

ラ

イ

ア

ン

ト

認

証

を

サ

ポ

ー

す

る

た

め

の

公

開

鍵

暗

号

標準

#11

の

リ

フ

レ

ン

ス

API
明
ア
ラ
リス
実
装
で
す。
詳
細
に
つ
い
て
は、
[「Amazon
FreeRTOS
公
開
鍵
暗
号
標
準
\(PKCS\)
#11
ライ
ブ
ラ
リ \(p. 166\)」](#)
を
参
照
し
て
く
だ
さ
い。

説明
明
ブア
ラ
リスト

詳S
細
につ
いて
は、
「Amazon
FreeRTOS Transport
Layer
Security
(TLS)
(p. 173)」
を参
照して
ください。

Amazon FreeRTOS アプリケーションライブラリ

AWS IoT を操作するために、以下のスタンドアロンアプリケーションライブラリを Amazon FreeRTOS 設定にオプションで含めることができます。

Amazon FreeRTOS アプリケーションライブラリ

説明
明
ブア
ラ
リスト

Greengrass
FreeRTOS
AWS
IoT
Greengrass
ラ
ズ
ブ
ラ
リ
は、Amazon

API
明
ア
ラ
ス

FreeRTOS

デ
バ
イ
ス
を

AWS
IoT
Greengrass

に
接
続
し
ま
す。

詳
細
に
つ
い
て
は、

「[Amazon
FreeRTOS
AWS
IoT
Greengrass
検出
ライブ
ラ
リ](#) (p. 146)」

を
参
照
し
て
く
だ
さ
い。

概要
明細
プラットフォーム
ラジオ
スイッチ

MQTT

Amazon FreeRTOS

MQTT

アーキテクチャ

オブジェクト

プロトコル

API

リソース

は、Amazon

FreeRTOS

デバイス

（ルータ）

スイッチ

MQTT

MQTT

ペッジ

エンド

を

ペア

API

リッスン

シミュレーター

、

オブジェクト

（スマートガラ

ン）

MQTT

API

する

ア

ため

の

データ

データ

アント

を

提供

しま

す。MQTT

は、

API
明
ア
ラ
ス
デ
バ
イ
ス
が
AWS
IoT
を
操
作
す
る
た
め
に
使
用
す
る
プロ
ト
コ
ル
で
す。

従
来
の
Amazon
FreeRTOS
MQTT

ライ
ブ
ラ
リ
の
詳
細
に
つ
い
て
は、
「[Amazon
FreeRTOS
MQTT
ラ](#)

API
明
ア
ラ
ス
イ
ブ
ラ
リ
(レ
ガ
シ
) (p. 161)」
を
参
照
して
く
だ
さ
い。

新
し
い
Amazon
FreeRTOS
MQTT
ラ
イ
ブ
ラ
リ
の
詳
細
に
つ
い
て
は、
パ
ブ
リ
ッ
ク
ベ
タ
で、
「[Amazon](#)
[FreeRTOS](#)
[MQTT](#)
ラ
イ
ブ
ラ

翻訳
朝
ブ
ラ
ウ
ス

リ
(ベー
タ) (p. 158)」

を
参
照
して
く
だ
さ
い。

API
明
ア
ラ
ス

AWS
Se

IoTShadow

API

バ

ア

ソ

シ

木

ウ

ライ

ブ

ラ

リ

は、Amazon
FreeRTOS

デ

バ

イ

ス

が

AWS

IoT

デ

バ

イ

ス

シャ

ド

ウ

を操

作

能

で

き

るよ

うにし

ま

す。

詳

細

に

つ

い

ては、

索引
明
ブ
ラ
ウ
ス

「Amazon
FreeRTOS
AWS
IoT
デ
バ
イ
ス
シ
ド
ウ
ラ
イ
ブ
ラ
リ (p. 144)」

を
参
照
して
く
だ
さ
い。

説明
アーキテクチャ
ラジオ
リスト

Amazon
FreeRTOS

AWS
IoT
Device
Defender

ライブ
ラリ

は、Amazon
FreeRTOS

デバ
イス
を

AWS
IoT
Device
Defender

に
接
続
し
ま
す。

詳細
につ
いて
は、

「Amazon

FreeRTOS

AWS

IoT

Device

Defender

ライ
ブ
ラ

リ (p. 140)

を
参
照

翻訳
朝
ア
ラ
ス
して
く
だ
さ
い。

Amazon FreeRTOS AWS IoT Device Defender ライ ブライ

概要

AWS IoT Device Defender は、デバイスの設定の監査、異常動作の検出を目的とした接続デバイスのモニタリング、セキュリティリスクの軽減を行う AWS IoT サービスです。また、このサービスでは、AWS IoT デバイスのフリートで IoT の設定を維持し、デバイスが侵害された場合にはすばやく応答することができます。

Amazon FreeRTOS には、Amazon FreeRTOS ベースのデバイスが AWS IoT Device Defender と連携するためのライブラリが用意されています。Amazon FreeRTOS コンソールを使用して Amazon FreeRTOS Device Defender ライブラリをダウンロードするには、Device Defender ライブラリをソフトウェア設定に追加します。また、Amazon FreeRTOS GitHub リポジトリをクローンし、lib ディレクトリでライブラリを見つけることもできます。

Amazon FreeRTOS AWS IoT Device Defender ライブライのソースファイルは、[AmazonFreeRTOS/lib/defender](#) にあります。

ソースとヘッダーファイル

```
Amazon FreeRTOS
| + - lib
|   | + - defender
|   |   + # aws_defender.c
|   |   + # aws_defender_states.dot
|   |   + # aws_defender_states.png
|   |   + # draw_states.py
|   |   + # portable
|   |       + # freertos
|   |           + # aws_defender_cpu.c
|   |           + # aws_defender_tcp_conn.c
|   |           + # aws_defender_uptime.c
|   |   + # stub
|   |       + # aws_defender_cpu.c
|   |       + # aws_defender_tcp_conn.c
|   |       + # aws_defender_uptime.c
|   |       + # makefile
```

```
+ # template
| + # aws_defender_cpu.c
| + # aws_defender_tcp_conn.c
| + # aws_defender_uptime.c
| + # makefile
+ # unit_test
| + # aws_defender_cpu.c
| + # aws_defender_tcp_conn.c
| + # aws_defender_uptime.c
+ # unix
| + # aws_defender_cpu.c
| + # aws_defender_tcp_conn.c
| + # aws_defender_uptime.c
| + # makefile
+ # report
| + # aws_defender_report.c
| + # aws_defender_report_cpu.c
| + # aws_defender_report_header.c
| + # aws_defender_report_tcp_conn.c
| + # aws_defender_report_uptime.c
+ - include
| + - aws_defender.h
| + - private
| | + - aws_defender_cpu.h
| | + - aws_defender_internals.h
| | + - aws_defender_report_cpu.h
| | + - aws_defender_report.h
| | + - aws_defender_report_header.h
| | + - aws_defender_report_tcp_conn.h
| | + - aws_defender_report_types.h
| | + - aws_defender_report_uptime.h
| | + - aws_defender_report_utils.h
| | + - aws_defender_tcp_conn.h
| | + - aws_defender_uptime.h
```

開発者サポート

Amazon FreeRTOS Device Defender API のエラーコード

eDefenderErrSuccess

オペレーションが成功しました。

eDefenderErrFailedToCreateTask

オペレーションを起動できませんでした。

eDefenderErrAlreadyStarted

オペレーションはすでに進行中です。

eDefenderErrNotStarted

Device Defender エージェントが起動していません。

eDefenderErrOther

不明なエラーが発生しました。

Amazon FreeRTOS Device Defender API

このセクションでは、Device Defender API についての情報が含まれています。

DEFENDER_MetricsInit

デバイスが AWS IoT Device Defender に送信する Device Defender メトリクスを指定します。

```
DefenderErr_t DEFENDER_MetricsInit(DefenderMetric_t * pxMetricsList);
```

引数

metrics_list

Device Defender メトリクスのリスト。有効な値は次のとおりです。

- DEFENDER_tcp_connections - TCP 接続の数を追跡します。

戻り値

DefenderErr_t 列挙値の 1 つを返します。詳細については、「[Amazon FreeRTOS Device Defender API のエラーコード \(p. 141\)](#)」を参照してください。

DEFENDER_ReportPeriodSet

レポート期間の間隔を秒単位で設定します。Device Defender は、特定の間隔でメトリクスレポートを提供します。デバイスが起動しており、その間隔が経過している場合、デバイスはメトリクスを報告します。

```
DefenderErr_t DEFENDER_ReportPeriodSet(int32_t LPeriodSec);
```

引数

period_sec

AWS IoT Device Defender にレポートを送信するまでの秒数。

戻り値

DefenderErr_t 列挙値の 1 つを返します。詳細については、「[Amazon FreeRTOS Device Defender API のエラーコード \(p. 141\)](#)」を参照してください。

DEFENDER_Start

Device Defender エージェントを開始します。

```
DefenderErr_t DEFENDER_Start(void);
```

戻り値

DefenderErr_t 列挙値の 1 つを返します。詳細については、「[Amazon FreeRTOS Device Defender API のエラーコード \(p. 141\)](#)」を参照してください。

DEFENDER_Stop

Device Defender エージェントを停止します。

```
DefenderErr_t DEFENDER_Stop(void);
```

戻り値

DefenderErr_t 列挙値の 1 つを返します。詳細については、「[Amazon FreeRTOS Device Defender API のエラーコード \(p. 141\)](#)」を参照してください。

DEFENDER_ReportStatusGet

最後の Device Defender レポートのステータスを取得します。有効なステータスコード値は次のとおりです。

eDefenderRepSuccess

最後のレポートは正常に送信され、確認されました。

eDefenderRepInit

Device Defender は開始しましたが、レポートは送信されませんでした。

eDefenderRepRejected

最後のレポートは拒否されました。

eDefenderRepNoAck

最後のレポートが認識されませんでした。

eDefenderRepNotSent

最後のレポートが送信されませんでした。接続の問題が原因の可能性があります。

```
DefenderReportStatus_t DEFENDER_ReportStatusGet(void);
```

使用例

組み込みアプリケーションでの Device Defender の使用

次のコードは、組み込みアプリケーションから Device Defender エージェントを設定および起動する方法を示しています。

```
void MyDefenderInit(void)
{
    // Specify metrics to send to Device Defender
    defender_metric_t metrics_list[] = {
        DEFENDER_tcp_connections
    };
    ( void ) DEFENDER_MetricsInit( metrics_list );

    // Set the reporting interval
    // You can use a shorter period to trigger the violation faster, however
    // the Device Defender service is not guaranteed to accept reports faster
```

```
// than every 300 seconds (5 minutes) per device.  
int report_period_sec = 300;  
( void ) DEFENDER_ReportPeriodSet( report_period_sec );  
  
// Start the Device Defender agent  
DEFENDER_Start();  
}
```

Amazon FreeRTOS AWS IoT デバイスシャドウライブラリ

概要

Amazon FreeRTOS Device Shadow API は、デバイスシャドウを作成、更新、削除する機能を定義しています。Amazon FreeRTOS デバイスのシャドウについての詳細は、「[デバイスシャドウ](#)」を参照してください。デバイスシャドウは、MQTT プロトコルを使用してアクセスできます。FreeRTOS Device Shadow API は、MQTT API と連携し、MQTT プロトコルの動作の詳細を処理します。

Amazon FreeRTOS AWS IoT デバイスシャドウライブラリのソースファイルは、[AmazonFreeRTOS/lib/shadow](#) にあります。

依存関係と要件

Amazon FreeRTOS で AWS IoT デバイスシャドウを使用するには、AWS IoT で証明書やポリシーなどのモノを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。AmazonFreeRTOS/demos/common/include/aws_client_credentials.h ファイルで、次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  
ネットワークで使用されている Wi-Fi セキュリティの種類。  
keyCLIENT_CERTIFICATE_PEM  
IoT モノに関連付けられた PEM 証明書。  
keyCLIENT_PRIVATE_KEY_PEM  
IoT モノに関連付けられているプライベートキー PEM。
```

お使いのデバイスに Amazon FreeRTOS MQTT ライブラリがインストールされていることを確認してください。詳細については、「[Amazon FreeRTOS MQTT ライブラリ \(レガシー\) \(p. 161\)](#)」を参照してください。

さい。MQTT バッファがシャドウ JSON ファイルを格納するのに十分な容量であることを確認してください。デバイスシャドウドキュメントの最大サイズは 8 KB です。Device Shadow API のすべてのデフォルト設定は、lib\include\private\aws_shadow_config_defaults.h ファイルで設定できます。これらの設定は、demos/<platform>/common/config_files/aws_shadow_config.h で変更できます。

AWS IoT に登録されている IoT モノと、デバイスシャドウへのアクセスを許可するポリシーを持つ証明書が必要です。詳細については、「[AWS IoT の使用開始](#)」を参照してください。

ソースとヘッダーファイル

```
Amazon FreeRTOS
+ - lib
|   + - shadow
|   |   + - aws_shadow.c
|   |   + - aws_shadow_json.c
|   + - include
|   |   + - aws_shadow.h
|   |   + - private
|   |   |   + - aws_shadow_config_defaults.h
|   |   |   + - aws_shadow_json.h
```

API リファレンス

完全な API リファレンスについては、「[Device Shadow API リファレンス](#)」を参照してください。

使用例

- SHADOW_ClientCreate API を使用してシャドウクライアントを作成します。ほとんどのアプリケーションでは、入力できる唯一のフィールドは xCreateParams.xMQTTClientType = eDedicatedMQTTClient です。
- SHADOW_ClientConnect API を呼び出し、SHADOW_ClientCreate で返されたクライアントハンドルを渡すことで MQTT 接続を確立します。
- SHADOW_RegisterCallbacks API を呼び出して、シャドウの更新、取得、および削除のコールバックを構成します。

接続が確立されたら、次の API を使用してデバイスシャドウを操作できます。

`SHADOW_Delete`

デバイスシャドウを削除します。

`SHADOW_Get`

現在のデバイスシャドウを取得します。

`SHADOW_Update`

デバイスシャドウを更新します。

Note

デバイスシャドウの操作が完了したら、SHADOW_ClientDisconnect を呼び出してシャドウクライアントとシステムリソースを解放します。

Amazon FreeRTOS AWS IoT Greengrass 検出ライブラリ

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。

Amazon FreeRTOS AWS IoT Greengrass ライブラリのソースファイルは、[AmazonFreeRTOS/lib/greengrass](#) にあります。

依存関係と要件

Greengrass Discovery ライブラリを使用するには AWS IoT で証明書とポリシーを含むモノを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。AmazonFreeRTOS \demos\common\include\aws_client_credentials.h ファイルで、次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。

`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、「[AWS IoT Greengrass の開始方法](#)」を参照してください。

MQTT 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。

ソースとヘッダーファイル

Amazon FreeRTOS

```
|  
+ - lib  
  + - greengrass  
    |  + # aws_greengrass_discovery.c  
    |  + # aws_helper_secure_connect.c  
  + - include  
    + - aws_greengrass_discovery.h  
    + - private  
      + - aws_ggd_config_defaults.h
```

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 ファイルを AWS IoT に要求することによって、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 lib\include\private\aws_ggd_config_defaults.h で定義されています。lib\include\ でこれらの設定を上書きすることができます。

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 が返されるごとに、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 コアを検出するための検出要求を開始するために、AWS IoT に対して HTTP GET 要求を行います。GD_SecureConnect_Send は AWS IoT に要求を送信するために使用されます。

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

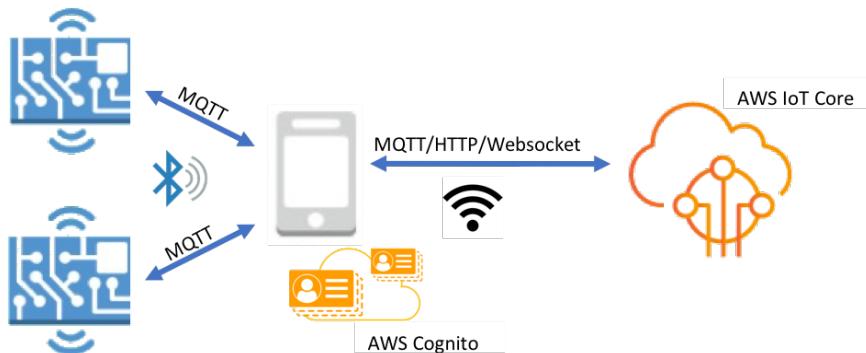
接続している Greengrass コアの ARN。

Amazon FreeRTOS Bluetooth Low Energy Library (ベータ)

概要

新しい Bluetooth Low Energy (BLE) ライブラリは、Amazon FreeRTOS パブリックベータリリースであり、変更される可能性があります。

Amazon FreeRTOS は、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 トピックのパブリッシュおよびサブスクライブをサポートします。Amazon FreeRTOS 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。

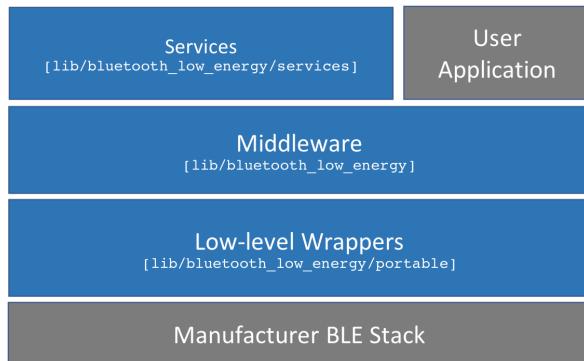


Amazon FreeRTOS Bluetooth デバイス用 Mobile SDK を使用すると、BLE を使用してマイクロコントローラの組み込みアプリケーションと通信するネイティブモバイルアプリケーションを記述できます。この Mobile SDK の詳細については、「[Amazon FreeRTOS Bluetooth デバイス用の Mobile SDK \(p. 157\)](#)」を参照してください。

Amazon FreeRTOS BLE ライブラリは、MQTT をサポートするだけでなく、Wi-Fi ネットワークを設定するためのサービスも含まれています。Amazon 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。Amazon FreeRTOS BLE ライブラリのソースファイルは、[AmazonFreeRTOS/lib/bluetooth_low_energy](#) にあります。

Amazon FreeRTOS BLE アーキテクチャー

Amazon FreeRTOS BLE ライブラリは、サービス、ミドルウェア、および低レベルのラッパーの 3 つのレイヤーで構成されています。



サービス

Amazon FreeRTOS BLE サービスレイヤーは、ミドルウェア API であるデバイス情報、Wi-Fi プロビジョニング、および BLE に対する MQTT 通信を活用する 3 つの汎用属性 (GATT) サービスで構成されています。詳細については、「[サービス \(p. 150\)](#)」を参照してください。

ミドルウェア

Amazon FreeRTOS BLE ミドルウェアは、下位レベルの API からの抽象化です。ミドルウェア API は、BLE 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。詳細については、「[ミドルウェア \(p. 150\)](#)」を参照してください。

低レベルのラッパー

低レベルの Amazon FreeRTOS BLE ラッパーは、製造元の BLE 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。Amazon FreeRTOS BLE サービスを使用するには、低レベルの API よりも多くのリソースを必要とする BLE サービス API を操作します。

依存関係と要件

MQTT over BLE および Wi-Fi プロビジョニングサービスのみに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
MQTT over BLE	Amazon FreeRTOS MQTT ライブラリ (ベータ) (p. 158)
Wi-Fi プロビジョニング	Amazon FreeRTOS Wi-Fi ライブラリ (p. 173)

AWS IoT MQTT ブローカーと通信するには、AWS アカウントが必要で、デバイスを AWS IoT 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。

Amazon FreeRTOS BLE 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 ID とユーザー プールを作成する必要があります。各 Amazon Cognito Identity 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 開発者ガイド](#)」を参照してください。

機能

サービス

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情報を収集します。

- 使用している Amazon 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。
- BLE 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。

MQTT over BLE

MQTT over BLE サービスは、マイクロコントローラーを Bluetooth 対応モバイルデバイスに接続し、AWS Mobile SDK を使用して AWS IoT クラウドに間接的に接続します。マイクロコントローラーは、MQTT クライアント、MQTT プロキシとしてのモバイルデバイス、MQTT サーバーとしての AWS IoT クラウドとして機能します。

ミドルウェア

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。

柔軟なコールバックサブスクリプション

BLE ハードウェアが切断され、MQTT over BLE サービスが切断を検出する必要があるとします。あなたが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。BLE 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。

ソースとヘッダーファイル

以下のツリーダイアグラムは、必要なソースファイルとヘッダーファイル、および Amazon FreeRTOS ディレクトリ構造内の場所を示しています。プロジェクトは、依存ライブラリのソースファイルもビルドする必要があります。

```
Amazon FreeRTOS
  |
  + - lib
    + - bluetooth_low_energy
      |   + - aws_ble_event_manager.c
      |   + - aws_ble_gap.c      [Middleware GAP]
      |   + - aws_ble_gatt.c     [Middleware GATT]
```

```
+ - portable      [Wrappers, wrapping APIs in lib/include/bluetooth_low_energy]
+ - services
    + - device_information [Service providing device info to the phone APP]
        | + - aws_ble_device_information.c
        + - mqtt_ble           [Used to do MQTT over BLE]
        | + - aws_mqtt_proxy.c
        + - wifi_provisioning   [WIFI provisioning service over BLE]
        | + - aws_ble_wifi_provisioning.c
+ - include
    + - bluetooth_low_energy [Wrapping APIs in lib/include/bluetooth_low_energy]
    | + - bt_hal_avsrc_profile.h
    + # bt_hal_gatt_client.h
    + # bt_hal_gatt_server.h
    + # bt_hal_gatt_types.h
    + # bt_hal_manager_adapter_ble.h
    + # bt_hal_manager_adapter_classic.h
    + # bt_hal_manager.h
    + # bt_hal_manager_types.h
    + - private            [For internal library use only!]
        | + - aws_ble_internals.h
        | + - aws_ble_config_defaults.h
        | + - aws_ble_event_manager.h
    + - aws_ble.h
    + - aws_ble_device_information.h
    + - aws_ble_services_init.h
    + - aws_ble_wifi_provisioning.h
```

Amazon FreeRTOSBLE ライブライアリ設定ファイル

Amazon FreeRTOS MQTT over BLE 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 `aws_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、`lib\include\private\aws_ble_config_defaults.h` で指定されたデフォルト値を使用します。

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。
- `aws_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。
- RAM を節約するために、未使用的サービスを削除することができます。
- 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、Amazon FreeRTOS BLE ライブライアリは `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。

より低い LE セキュリティレベルを使用するには、`eBTpropertySecureConnectionOnly` プロパティで API `pxSetDeviceProperty` を呼び出して `eBTpropertySecureConnectionOnly` を FALSE に設定します。

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 BLE 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

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。

ミドルウェアを初期化するには

1. BLE ミドルウェア API を呼び出す前に、BLE ハードウェアドライバを初期化する必要があります。
2. BLE を有効化します。

```
const BTInterface_t * pxIface = BTGetBluetoothInterface();
xStatus = pxIface->pxEnable( 0 );
```

3. BLE を初期化するには、セキュアな接続モード、デバイス名、MTU サイズなど、必要なプロパティのセットとともに、`BLE_Init` を呼び出します。

```
xStatus = BLE_Init( &xServerUUID, xDeviceProperties, MAX_PROPERTIES );
```

低レベル API

Amazon FreeRTOS BLE 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することができます。

低レベル API を初期化するには

1. ドライバの初期化は、BLE 低レベル API の一部ではありません。API を呼び出す前に、BLE ハードウェアドライバを初期化する必要があります。
2. BLE 低レベル API は、能力とリソースを最適化するために BLE 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 BLE を有効にする必要があります。

```
const BTInterface_t * pxIface = BTGetBluetoothInterface();
xStatus = pxIface->pxEnable( 0 );
```

3. Bluetooth マネージャには、BLE 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. BLE 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 );
```

BLE 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 \(BLE\) API リファレンス](#)」を参照してください。

使用例

広告

1. 広告パラメータを設定します。

```
BLEAdvertisementParams_t xAdvParams =
{
    .bIncludeTxPower      = true,
    .bIncludeName         = true,
    .bSetScanRsp          = true,
    .ulAppearance          = 0,
    .ulMinInterval        = 0x20,
    .ulMaxInterval        = 0x40,
    .usManufacturerLen   = 0,
    .pcManufacturerData  = NULL,
    .pxUUID1              = &xDeviceInfoSvcUUID,
    .pxUUID2              = NULL
};

if( xStatus == eBTStatusSuccess )
{
    ( void ) BLE_SetAdvData( BTAdvInd, &xAdvParams, vSetAdvCallback );
}
```

2. 広告を開始します。

```
void vSetAdvCallback ( BTStatus_t xStatus )
{
```

```
    if( xStatus == eBTStatusSuccess )
    {
        ( void ) BLE_StartAdv( vStartAdvCallback );
    }
}
```

新しいサービスの追加

- 新しいサービスのメモリを割り当てます。

```
xStatus = BLE_CreateService( &pxGattDemoService, gattDemoNUM_CHARS,
                                gattDemoNUM_CHAR_DESCRS, xNumDescrsPerChar, gattDemoNUM_INCLUDED_SERVICES );
```

- サービスを作成します。

```
pxGattDemoService->xAttributeData.xUuid = xServiceUUID;

pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xAttributeData.xUuid
    = xClientCharCfgUUID;
pxGattDemoService-
>pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xAttributeData.pucData = NULL;
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xAttributeData.xSize
    = 0;
pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ].xPermissions =
    ( eBTPermReadEncryptedMitm | eBTPermWriteEncryptedMitm );
pxGattDemoService-
>pxDescriptors[ egattDemoCharCounterCCFGDESCR ].pxAttributeEventCallback =
    vEnableNotification;

xCharUUID.uu.uu16 = gattDemoCHAR_COUNTER_UUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xAttributeData.xUuid =
    xCharUUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xAttributeData.pucData =
    NULL;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xAttributeData.xSize = 0;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xPermissions =
    ( eBTPermRead );
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xProperties =
    ( eBTPropRead | eBTPropNotify );
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].pxAttributeEventCallback =
    vReadCounter;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].xNbDescriptors = 1;
pxGattDemoService->pxCharacteristics[ egattDemoCharCounter ].pxDescriptors[ 0 ] =
    &pxGattDemoService->pxDescriptors[ egattDemoCharCounterCCFGDESCR ];

xCharUUID.uu.uu16 = gattDemoCHAR_CONTROL_UUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xAttributeData.xUuid =
    xCharUUID;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xAttributeData.pucData =
    NULL;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xAttributeData.xSize = 0;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xPermissions =
    ( eBTPermReadEncryptedMitm | eBTPermWriteEncryptedMitm );
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xProperties =
    ( eBTPropRead | eBTPropWrite );
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].pxAttributeEventCallback =
    vWriteCommand;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].xNbDescriptors = 0;
pxGattDemoService->pxCharacteristics[ egattDemoCharControl ].pxDescriptors = NULL;

pxGattDemoService->xServiceType = eBTServiceTypePrimary;
pxGattDemoService->ucInstId = 0;
```

```
xStatus = BLE_AddService( pxGattDemoService );
```

3. サービスを起動します。

```
xStatus = BLE_StartService( pxGattDemoService, vServiceStartedCb );
```

4. サービスに必要なすべてのイベントをサブスクライブします。この例では、イベントの接続をサブスクライブします。

```
xCallback.pxConnectionCb = vConnectionCallback;  
BLE_RegisterEventCb( eBLEConnection, xCallback );
```

完全な Amazon FreeRTOS BLE デモアプリケーションについては、「[Bluetooth Low Energy デモアプリケーション](#)」を参照してください。

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

Amazon 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 BLE 機能用にいくつかの API を実装する必要があります。

Wi-Fi プロビジョニング API

Wi-Fi プロビジョニングサービスを使用するには、次の API を実装する必要があります。

- WIFI_NetworkGet
- WIFI_NetworkDelete
- WIFI_NetworkAdd

BLE API

Amazon FreeRTOS BLE 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 BLE 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`

BLE 用 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

Amazon FreeRTOS Bluetooth デバイス用の Mobile SDK

新しい Bluetooth Low Energy (BLE) ライブラリは、Amazon FreeRTOS パブリックベータリリースであり、変更される可能性があります。

Amazon FreeRTOS Bluetooth デバイス用 Mobile SDK を使用して、BLE 経由でマイクロコントローラを操作するモバイルアプリケーションを作成できます。また、この Mobile SDK は、ユーザー認証用の Amazon Cognito を使用して、AWS のサービスと通信することもできます。

Amazon FreeRTOS Bluetooth デバイス用の Android SDK

Amazon FreeRTOS Bluetooth デバイス用 Android SDK を使用して、BLE 経由でマイクロコントローラを操作する Android モバイルアプリケーションをビルドします。SDK は [GitHub](#) で入手できます。

Android SDK をインストールするには

1. SDK を [GitHub](#) からダウンロードします。
2. Android Studio を開き、`amazon-freertos-ble-android-sdk/amazonfreertosdk/` ディレクトリをアプリプロジェクトにインポートします。
3. アプリの `gradle` ファイルに、次の依存関係を追加します。

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

4. アプリの `AndroidManifest.xml` ファイルに、次のアクセス許可を追加します。

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

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

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 \(p. 183\)](#)」および「[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#)」を参照してください。

Amazon FreeRTOS Bluetooth デバイス用の iOS SDK

Amazon FreeRTOS Bluetooth デバイス用 iOS SDK を使用して、BLE 経由でマイクロコントローラを操作する iOS モバイルアプリケーションをビルドします。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 で SDK をインストールする:

```
$ pod 'AmazonFreeRTOS', :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 \(p. 183\)](#)」および「[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#)」を参照してください。

Amazon FreeRTOS MQTT ライブラリ (ベータ)

新しい MQTT ライブラリは、Amazon FreeRTOS パブリックベータリリースであり、変更される可能性があります。

概要

Amazon FreeRTOS MQTT ライブラリを使用し、ネットワークの MQTT クライアントとして、MQTT トピックをパブリッシュおよびサブスクライブするアプリケーションを作成できます。Amazon FreeRTOS

MQTT ライブリは、AWS IoT MQTT サーバーとの互換性のために MQTT 3.1.1 規格を実装しています。ライブラリは他の MQTT サーバーと互換性があります。

Amazon FreeRTOS MQTT ライブリのソースファイルは、[AmazonFreeRTOS/lib/mqtt](#) にあります。

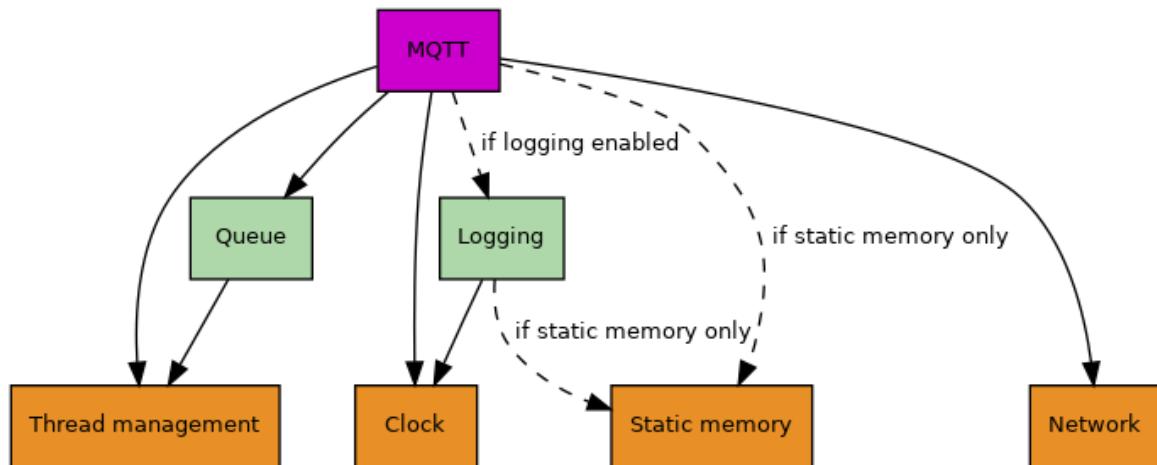
ここに記載されている Amazon FreeRTOS MQTT ライブリはパブリックベータにあります。従来の Amazon FreeRTOS MQTT ライブリの詳細については、「[Amazon FreeRTOS MQTT ライブリ \(レガシー\) \(p. 161\)](#)」を参照してください。

依存関係と要件

Amazon FreeRTOS MQTT ライブリには、以下の依存関係があります。

- 進行中の MQTT オペレーションを管理するデータ構造を維持するためのキューライブリ
- ログ記録のライブリ (設定パラメータ AWS_IOT_MQTT_LOG_LEVEL が AWS_IOT_LOG_NONE に設定されていない場合)
- スレッド管理、クロック関数、ネットワーク、およびその他のプラットフォームレベルの機能のために、オペレーティングシステムへのインターフェイスを提供するプラットフォームレイヤー
- C 標準ライブリヘッダー

下の図は、これらの依存関係を示しています。



機能

Amazon FreeRTOS MQTT ライブリには、以下の機能があります。

- デフォルトでは、ライブリには完全非同期 MQTT API があります。ライブリを `AwsIotMqtt_Wait` 関数と同期して使用することができます。
- ライブリは、高いスループットのためにスレッドを認識し、並列化できます。
- ライブリはスケーラブルなパフォーマンスとフットプリントを備えています。構成設定を使用して、ライブリをシステムのリソースに合わせて調整します。

設定

Amazon FreeRTOS MQTT ライブリの構成設定は、C プリプロセッサ定数として定義されています。構成設定を `AWS_IOT_CONFIG_FILE` という名前のファイルに `#define` 定数として設定するか、`gcc` の `-D` などのコンパイラオプションを使用して設定します。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。構成設定が定義されていない場合は、MQTT ライブラリはデフォルト値を使用します。

Amazon FreeRTOS MQTT ライブラリの設定の詳細については、「[MQTT API リファレンス \(ベータ\)](#)」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、「[MQTT API リファレンス \(ベータ\)](#)」を参照してください。

使用例

`aws_iot_demo_mqtt.c`

Amazon FreeRTOS MQTT ライブラリの使用例については、`aws_iot_demo_mqtt.c` に定義されている MQTT デモアプリケーションを参照してください。

MQTT デモは、MQTT のサブスクライブおよびパブリッシュワークフローを示しています。複数のトピックフィルターをサブスクライブした後で、アプリケーションはさまざまなトピック名にデータのバーストをパブリッシュします。各メッセージが到着すると、デモは受信確認メッセージを MQTT サーバーに戻します。

MQTT デモを実行するには、以下のパラメータを設定する必要があります。

グローバルデモ設定パラメータ

これらの設定パラメータはすべてのデモに適用されます。

`AWS_IOT_DEMO_SECURED_CONNECTION`

デモがデフォルトでリモートホストとの TLS セキュリティで保護された接続を使用するかどうかを決定します。

`AWS_IOT_DEMO_SERVER`

使用するデフォルトのリモートホスト。

`AWS_IOT_DEMO_PORT`

使用するデフォルトのリモートポート。

`AWS_IOT_DEMO_ROOT_CA`

使用するデフォルトの信頼されたサーバールート証明書へのパス。

`AWS_IOT_DEMO_CLIENT_CERT`

使用するデフォルトのクライアントサーバールート証明書へのパス。

`AWS_IOT_DEMO_PRIVATE_KEY`

使用するデフォルトのクライアント証明書プライベートキーへのパス。

MQTT デモ設定パラメータ

これらの設定パラメータは MQTT デモに適用されます。

`AWS_IOT_DEMO_MQTT_PUBLISH_BURST_SIZE`

各バーストでパブリッシュするメッセージの数。

AWS_IOT_DEMO_MQTT_PUBLISH_BURST_COUNT

このデモのパブリッシュバーストの数。

Amazon FreeRTOS MQTT ライブラリ (レガシー)

概要

Amazon FreeRTOS には、オープンソースの MQTT クライアントライブラリが含まれています。このライブラリを使用して、MQTT トピックをパブリッシュおよびサブスクライブするアプリケーションをネットワーク上の MQTT クライアントとして作成できます。

Amazon FreeRTOS MQTT ライブラリのソースファイルは、[AmazonFreeRTOS/lib/mqtt](#) にあります。

新しい Amazon FreeRTOS MQTT ライブラリはパブリックベータにあります。詳細については、「[Amazon FreeRTOS MQTT ライブラリ \(ベータ\) \(p. 158\)](#)」を参照してください。

FreeRTOS MQTT エージェント

Amazon FreeRTOS には、MQTT ライブラリを管理する FreeRTOS MQTT エージェントと呼ばれるオープンソースデーモンも含まれています。MQTT エージェントは、基盤となる MQTT ライブラリを使用して MQTT トピックの接続、パブリッシュ、およびサブスクライブを行うためのシンプルなインターフェイスを提供します。

MQTT エージェントは個別の FreeRTOS タスクで実行され、MQTT プロトコル仕様に記載されているように、通常のキープアライブメッセージを自動的に送信します。すべての MQTT API はプロックされており、対応する操作が完了するまで API が待機する最大時間であるタイムアウトパラメーターを使用します。指定された時間内に操作が完了しない場合、API はタイムアウトエラーコードを返します。

依存関係と要件

Amazon FreeRTOS MQTT ライブラリは、[Amazon FreeRTOS セキュアソケットライブラリ \(p. 168\)](#) と [Amazon FreeRTOS バッファプールライブラリ](#)を使用します。MQTT エージェントがセキュアな MQTT ブローカーに接続する場合、ライブラリは [Amazon FreeRTOS Transport Layer Security \(TLS\) \(p. 173\)](#) も使用します。

機能

コールバック

MQTT エージェントがブローカーから切断されるたびに、またはブローカーからパブリッシュメッセージを受信するたびに呼び出されるオプションのコールバックを指定できます。受信されたパブリッシュメッセージは、中央バッファプールから取られたバッファに格納されます。このメッセージはコールバックに渡されます。このコールバックは MQTT タスクのコンテキストで実行されるため、高速でなければなりません。より長い処理が必要な場合は、コールバックから pdTRUE を返してバッファの所有権を取得する必要があります。FreeRTOS_Agent_ReturnBuffer を呼び出してバッファをプールに戻す必要があります。

サブスクリプションの管理

サブスクリプションの管理では、サブスクリプションフィルタごとにコールバックを登録できます。サブスクライブにこのコールバックを指定します。トピックで受信したパブリッシュメッセージが、サブスクライブしたトピックフィルターと一致するたびに呼び出されます。バッファの所有権は、一般的なコールバックの場合と同じように機能します。

MQTT タスクウェイクアップ

MQTT タスクウェイクアップは、ユーザーが API を呼び出して操作を実行するたびに、またはパブリッシュメッセージがブローカーから受信されるたびに起動します。このパブリッシュメッセージを受信する時の非同期のウェイクアップは、接続されたソケットで受信したデータについてホスト MCU に通知できるプラットフォームで可能です。この機能を持たないプラットフォームでは、MQTT タスクは、接続されたソケット上の受信データを継続的にポーリングする必要があります。パブリッシュメッセージを受信してからコールバックを呼び出すまでの遅延が最小限になるように、`mqttconfigMQTT_TASK_MAX_BLOCK_TICKS` マクロは、MQTT タスクがブロックされたままになる最大時間を制御します。接続されたソケット上の受信データをホスト MCU に通知する能力が不足しているプラットフォームでは、この値を小さくする必要があります。

ソースとヘッダーファイル

```
Amazon FreeRTOS
  |
  + - lib
    |
    + - mqtt
      |   + - aws_mqtt_lib.c           [Required to use the MQTT library and the
      |   |   + - aws_mqtt_agent.c       [Required to use the MQTT agent]
      |   |
      |   + - include
      |     |
      |     + - private               [For internal library use only!]
      |       + - aws_doubly_linked_list.h
      |       +- aws_mqtt_agent_config_defaults.h
      |       + - aws_mqtt_buffer.h
      |       + - aws_mqtt_config_defaults.h
      |
      + - aws_mqtt_agent.h         [Include to use the MQTT agent API]
      + - aws_mqtt_lib.h          [Include to use the MQTT library API]
```

主な設定

次のフラグは、MQTT 接続要求中に指定することができます。

- `mqttconfigKEEP_ALIVE_ACTUAL_INTERVAL_TICKS`: 送信されるキープアライブメッセージの頻度。
- `mqttconfigENABLE_SUBSCRIPTION_MANAGEMENT`: サブスクリプションの管理。
- `mqttconfigMAX_BROKERS`: 同時 MQTT クライアントの最大数。
- `mqttconfigMQTT_TASK_STACK_DEPTH`: タスクスタックの深度。
- `mqttconfigMQTT_TASK_PRIORITY`: MQTT タスクの優先順位。
- `mqttconfigRX_BUFFER_SIZE`: データを受信するために使用されるバッファの長さ。
- `mqttagentURL_IS_IP_ADDRESS`: 提供された URL が IP アドレスの場合、`xFlags` でこのビットを設定します。
- `mqttagentREQUIRE_TLS`: このビットを `xFlags` に設定すると、TLS が使用されます。
- `mqttagentUSE_AWS_IOT_ALPN_443`: このビットを `xFlags` に設定すると、TLS ポート 443 で MQTT の AWS IoT サポートが使用されます。

ALPN の詳細については、「AWS IoT 開発者ガイド」の「[AWS IoT プロトコル](#)」および AWS IoT ブログにある「[ポート 443 で TLS 認証を使った MQTT: なぜ便利で、どのように動くのか](#)」を参照してください。

最適化

遅滞なく受信パケットを処理する

MQTT エージェントを実装するタスクは、イベントが処理されるのを待機しているブロック状態 (CPU サイクルを使用しないため) でほとんどの時間を費やします。MQTT パケットがネットワークから受信されるとただちにエージェントタスクのブロックを解除することにより、MQTT スループットが最大化されます。これが行われた場合、受信パケットはできるだけ早く処理されます。これが行われないと、MQTT エージェントが別の理由でブロック状態を離れるまで、受信パケットは処理されません。

MQTT エージェントは、IOptionName パラメータを SOCKETS_SO_WAKEUP_CALLBACK に設定して SOCKETS_SetSockOpt() を呼び出す MQTT エージェントによってインストールされるコールバックの実行によって、ブロック状態から削除されます。ここでは、セキュアソケットのドキュメントへのリンクが必要です。FreeRTOS+TCP TCP/IP スタックを使用している場合は、FreeRTOSIPConfig.h (TCP/IP スタックの設定ファイル) で ipconfigSOCKET_HAS_USER_WAKE_CALLBACK が 1 に設定されている場合、コールバックは正しい時刻に実行されます。FreeRTOS+TCP TCP/IP スタックを使用していない場合、セキュアソケットは、使用中のスタックのセキュアソケット抽象化レイヤーの実装にこの機能が含まれていることを保証します。

データが受信されてすぐに TCP/IP スタックが MQTT エージェントをブロック解除できない場合、受信されているパケットと処理されているパケットの間の最大時間は、mqttconfigMQTT_TASK_MAX_BLOCK_TICKS 定数によって設定されます。

RAM の消費を最小限に抑える

次の設定定数は、MQTT エージェントが必要とする RAM の量に直接影響します。

- mqttconfigMQTT_TASK_STACK_DEPTH
- mqttconfigMQTT_TASK_STACK_DEPTH
- mqttconfigMAX_BROKERS
- mqttconfigMAX_PARALLEL_OPS
- mqttconfigRX_BUFFER_SIZE

これらの定数は可能な最小値に設定する必要があります。

要件と使用制限

MQTT エージェントタスクは xTaskCreateStatic() API 関数を使用して作成されるため、タスクのスタックおよび制御ブロックはコンパイル時に静的に割り当てられます。これにより、動的メモリ割り当てを許可しないアプリケーションで MQTT エージェントを使用できるようになりますが、FreeRTOSConfig.h の configSUPPORT_STATIC_ALLOCATION に 1 が設定されていることに依存します。

MQTT エージェントは、FreeRTOS 直接通知タスク機能を使用します。MQTT エージェント API 関数を呼び出すと、呼び出し側のタスクの通知値と状態が変更されることがあります。

MQTT パケットは、バッファプールモジュールによって提供されるバッファに保存されます。プール内のバッファの数が、一度に進行中の MQTT トランザクションの数の 2 倍以上にすることを強く推奨します。

開発者サポート

mqttconfigASSERT

mqttconfigASSERT() は、FreeRTOS configASSERT() macro マクロと同等で、まったく同じ方法で使用されます。MQTT エージェントにアサーションステートメントを挿入する場合は、mqttconfigASSERT() を定義します。MQTT エージェントにアサーションステートメント

を必要としない場合は、`mqttconfigASSERT()` を未定義のままにしておきます。以下に示すように、`mqttconfigASSERT()` を定義して FreeRTOS configASSERT() を呼び出すと、FreeRTOS configASSERT() が定義されている場合にのみ、MQTT エージェントにアサーションステートメントが組み込まれます。

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

mqttconfigENABLE_DEBUG_LOGS

`vLoggingPrintf()` の呼び出しを介してデバッグログを出力するには、`mqttconfigENABLE_DEBUG_LOGS` を 1 に設定します。

初期化

MQTT 通信を試みる前に、以下に示すように、MQTT エージェントとその依存ライブラリの両方を初期化する必要があります。ネットワーク接続が確立されたら、ライブラリを初期化します。

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

API リファレンス

完全な API リファレンスについては、「[MQTT ライブラリ API リファレンス \(レガシー\)](#)」および「[MQTT エージェント API リファレンス \(レガシー\)](#)」を参照してください。

移植

MQTT エージェントが呼び出す Secure Sockets 抽象化レイヤーは、特定のアーキテクチャーに移植する必要があります。詳細については、「[Amazon FreeRTOS 移植ガイド](#)」を参照してください。

Amazon FreeRTOS 無線通信 (OTA) エージェント ライブラリ

概要

OTA エージェントを使用すると、Amazon FreeRTOS デバイスのファームウェアアップデートの通知、ダウンロード、および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、ファームウェアのアップデート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、または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。

Amazon FreeRTOS での無線通信経由の更新の設定の詳細については、「[Amazon FreeRTOS 無線による更新 \(p. 7\)](#)」を参照してください。

Amazon FreeRTOS OTA エージェントライブラリのソースファイルは、[AmazonFreeRTOS/lib/ota](#) にあります。

機能

OTA エージェントの各インターフェースは次のとおりです。

OTA_AgentInit

OTA エージェントを初期化します。呼び出し元は、メッセージングプロトコルコンテキスト、オプションのコールバック、およびタイムアウトを提供します。

OTA_AgentShutdown

OTA エージェントを使用した後にリソースをクリーンアップします。

OTA_GetAgentState

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。

ソースとヘッダーファイル

```
Amazon FreeRTOS
|
+ - lib
    + - ota
        + - aws_ota_agent.c
        + - aws_ota_cbor.c
        + - portable
            + - README.md
        + - vendor
            + - board
                + - aws_ota_pal.c
    + - include
        + - aws_ota_agent.h
        + - private
            + - aws_ota_agent_internal.h
            + - aws_ota_cbor.h
            + - aws_ota_cbor_internal.h
            + - aws_ota_pal.h
            + - aws_ota_types.h
```

API リファレンス

完全な API リファレンスについては、「[OTA エージェント API リファレンス](#)」を参照してください。

使用例

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 MQTT ブローカーに接続します。詳細については、「[Amazon FreeRTOS MQTT ライブラリ \(レガシー\) \(p. 161\)](#)」を参照してください。
 2. OTA_AgentInit 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。アプリケーションでカスタム OTA コールバック関数を定義するか、NULL コールバック関数ポインタを指定してデフォルトのコールバックを使用することができます。初期化タイムアウトも指定する必要があります。
- 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。タイムアウトは、初期化が完了するまでの待機時間を定義します。
3. エージェントの準備が完了する前に OTA_AgentInit がタイムアウトした場合は、OTA_GetAgentState を呼び出して、エージェントが初期化されていることを確認できます。
 4. OTA の更新が完了すると、Amazon 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コールします。
 5. 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。
 6. 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、「[OTA ポータブル抽象化レイヤー](#)」を参照してください。

Amazon FreeRTOS 公開鍵暗号標準 (PKCS) #11 ライブラリ

概要

公開鍵暗号標準 #11 (PKCS#11) 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する暗号化 API です。Amazon FreeRTOS のソースコードリポジトリの `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。Amazon 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作成されます。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一部のみを実装する必要があります。

Amazon FreeRTOS PKCS#11 ライブラリのソースファイルは、[AmazonFreeRTOS/lib/secure_sockets/portable](#) にあります。

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。

プロジェクトジョギング API

- C_GetFunctionList
- C_Initialize
- 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

クライアント認証

- C_Initialize
- C_GetSlotList
- C_OpenSession
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_GetAttributeValue
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_GetAttributeValue
- C_GenerateRandom
- C_SignInit
- C_Sign
- C_DigestInit
- C_DigestUpdate
- C_DigestFinal

クリーンアップ

- C_CloseSession
- C_Finalize

非対称暗号化方式のサポート

Amazon FreeRTOS PKCS#11 リファレンス実装では、NIST P-256 曲線で 2048 ビットの RSA (署名のみ) と ECDSA をサポートしています。以下の手順では、P-256 クライアント証明書に基づいて AWS IoT を作成する方法について説明します。

AWS CLI および OpenSSL の次のバージョン (またはそれより新しいバージョン) を使用していることを確認してください。

```
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
```

次のステップでは、aws configure コマンドを使用して AWS CLI を設定したことを前提としています。

P-256 クライアント証明書に基づく AWS IoT モノの作成

1. aws iot create-thing --thing-name dcgecc を実行して、AWS IoT モノを作成します。
2. openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -pkeyopt ec_param_enc:named_curve -outform PEM -out dcgecc.key を実行し、OpenSSL を使用して P-256 キーを作成します。
3. openssl req -new -nodes -days 365 -key dcgecc.key -out dcgecc.req を実行し、ステップ 2 で作成したキーで署名された証明書の登録要求を作成します。
4. aws iot create-certificate-from-csr --certificate-signing-request file://dcgecc.req --set-as-active --certificate-pem-outfile dcgecc.crt を実行し、証明書の登録要求を AWS IoT に送信します。
5. aws iot attach-thing-principal --thing-name dcgecc --principal "arn:aws:iot:us-east-1:123456789012:cert/86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de" を実行して証明書(前のコマンドで ARN 出力によって参照される)をモノにアタッチします。
6. aws iot create-policy --policy-name FullControl --policy-document file://policy.json を実行し、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。)

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 Amazon FreeRTOS の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. aws iot attach-principal-policy --policy-name FullControl --principal "arn:aws:iot:us-east-1:785484208847:cert/86e41339a6d1bbc67abf31faf455092cdebf8f21ffbc67c4d238d1326c7de" を実行し、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。

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。

Amazon FreeRTOS セキュアソケットライブラリ 概要

Amazon 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験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。

Amazon FreeRTOS セキュアソケットライブラリは、バーカ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。Amazon FreeRTOS セキュアソケットライブラリとバーカ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、[Secure Sockets API リファレンス](#) の「SOCKETS_SetSockOpt」を参照してください。

Amazon FreeRTOS セキュアソケットライブラリのソースファイルは、[AmazonFreeRTOS/lib/secure_sockets/portable](#) にあります。

Note

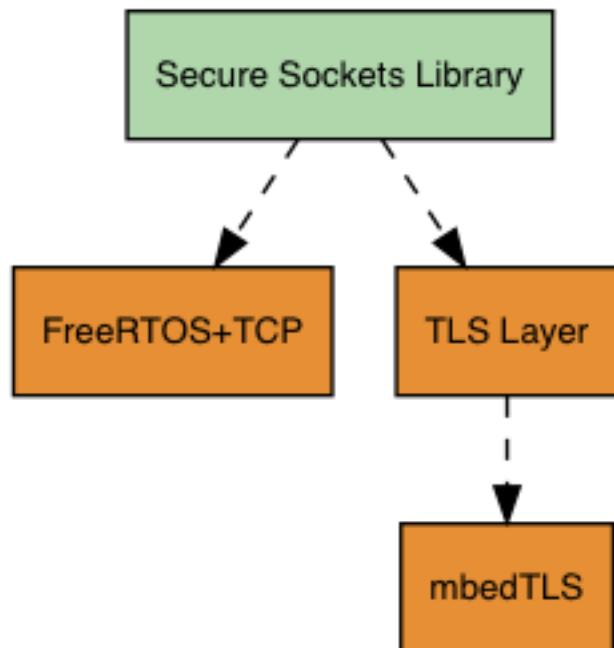
現在のところ、Amazon FreeRTOS セキュアソケットでサポートされているのはクライアント API のみです。

依存関係と要件

Amazon 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。Amazon 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 と、[mbedtls](#) による Amazon FreeRTOS TLS レイヤーのカスタム実装
- [FreeRTOS+TCP](#) と、[mbedtls](#) による Amazon FreeRTOS TLS レイヤー

以下の依存関係の図は、Amazon 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。Amazon FreeRTOS TLS レイヤーの詳細については、「[Amazon FreeRTOS Transport Layer Security \(TLS\) \(p. 173\)](#)」を参照してください。



機能

Amazon 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。

- 標準のバーカ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

フットプリント

コードサイズ (ARM Cortex-M 向けの GCC で生成された例)

ファイル名	サイズ (速度のために最適化)	サイズ (速度とサイズのために最適化)
セキュアソケットライブラリ	ポートによって異なる	ポートによって異なる
たとえば、TI CC3220SF の場合:	5.0 K	4.3 K
<code>lib/secure_sockets/portable/ti/cc3220_launchpad/aws_secure_sockets.c</code>		

ソースとヘッダーファイル

```
Amazon FreeRTOS
|
+ - lib
  + - include
    | + - aws_secure_sockets.h
    | + - private
    |   + - aws_secure_sockets_config_defaults.h
  + - secure_sockets
    + - portable
      + - ...
        + - aws_secure_sockets.c
```

トラブルシューティング

エラーコード

Amazon 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、[Secure Sockets API リファレンス](#) で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

Amazon FreeRTOS セキュアソケット API がエラーコードを返した場合、Amazon FreeRTOS セキュアソケットライブラリに依存している [Amazon FreeRTOS MQTT ライブラリ \(レガシー\) \(p. 161\)](#) 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

Amazon 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

Amazon 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。

Amazon 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クライアント API のみです。Bind、Accept、Listen など、サーバー API はサポートされていません。

初期化

Amazon 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 リファレンス](#)」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.\n
Change this to the certificate of your choice. */
static const char ctlsECHO_SERVER_CERTIFICATE_PEM[] =
"-----BEGIN CERTIFICATE-----\n"
"MIIBtjCCAVugAwIBAgITBmyf1XSXNmY/Owua2eiedgPySjAKBggqhkjOPQDAjA5\n"
```

```

"MQSwCQYDVQQGEwJVUzEPMA0GA1UEChMGQW1hem9uMRkwFwYDVQQDExBBbWF6b24g\n"
"Um9vdCBDQSAzMb4XDTE1MDUyNjAwMDAwMFoXDTQwMDUyNjAwMDAwMFowOTELMAkG\n"
"A1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJvb3Qg\n"
"Q0EgMzBZMBMGBYqGSM49AgEGCCqGSM49AwEHA0IABCmXp8ZBF8ANm+gBG1bG81Kl\n"
"ui2yEujSLtf6ycXYqm0fc4E7O5hrOXwzpcVHo6AF2hiRVd9RFgdszf1ZwjrZt6j\n"
"QjBAMA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/BAQDAgGGMB0GA1UdDgQWBBSr\n"
"ttvXBp43rDCGB5Fwx5zEGbF4wDAKBggqhkjOPQQDAGNJADBGAEA4IWSoxe3jfkr\n"
"BqWtrBqYagFy+uGh0PsceGCMq5nFuMQCIQCcAu/x1Jyz1vnrxir4tiz+OpAUFeM\n"
"YyRIHN8wfdfVoOw==\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_SNDFTIMEO, &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, &ampxEchoServerAddress, sizeof( xEchoServerAddress ) ) ==
0 )
    {
        /* Send the string to the socket. */
        xTransmitted = SOCKETS_Send( xSocket,                                     /* The socket
receiving. */
                                    ( void * )"some message",           /* The data being
sent. */                                /* The length of the
data being sent. */
                                    12,                                         /* No flags. */
                                    0 );
        if( xTransmitted < 0 )
        {
            /* Error while sending data*/
            return;
        }

        SOCKETS_Shutdown( xSocket, SOCKETS_SHUT_RDWR );
    }
    else
    {
        //failed to connect to server
    }
}

```

```
    SOCKETS_Close( xSocket );  
}
```

コード例の全体については、[Secure Sockets Echo Client Demo](#) を参照してください。

移植

Amazon 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
- [Amazon FreeRTOS 公開鍵暗号標準 \(PKCS\) #11 ライブラリ \(p. 166\)](#)
- [Amazon FreeRTOS Transport Layer Security \(TLS\) \(p. 173\)](#)

移植の詳細については、「[デバイスの移植 \(p. 204\)](#)」を参照してください。

Amazon FreeRTOS Transport Layer Security (TLS)

Amazon 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セキュアソケット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、セキュアソケット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。Amazon FreeRTOS ソースコードリポジトリの `aws_tls.h` を参照してください。

セキュアソケットから 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。

Amazon FreeRTOS Wi-Fi ライブラリ

概要

Amazon 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 Amazon 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。

Amazon FreeRTOS Wi-Fi ライブラリのソースファイルは、[AmazonFreeRTOS/lib/wifi/portable](#) にあります。

依存関係と要件

Amazon 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、および WPA2 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。

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、および WPA2 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`

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 \(p. 176\)](#)」および「[API リファレンス \(p. 176\)](#)」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_SetPMMMode 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_GetPMMMode 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。

フットプリント

コードサイズ (ARM Cortex-M 向けの GCC で生成された例)

ファイル名	サイズ (-O1 最適化)	サイズ (Os 最適化)
Wi-Fi ライブラリ、すべてのオプションが有効	ポートによって異なる	ポートによって異なる
たとえば、TI CC3220SF の場合: lib/ wifi/portable/ ti/ cc3220_launchpad/ aws_wifi.c	3.7 K	3.0 K

ソースとヘッダーファイル

```
Amazon FreeRTOS
|
+ - lib
    + - include
        |   + - aws_wifi.h           [Include to use the AFR WIFI API]
    + - wifi
        + - portable
            + ...
                + - aws_wifi.c       [Port-specific folder structure]
                                    [Required to use the AFR WIFI API]
```

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 \(p. 176\)](#)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 ライブライア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、[demos/vendor/board/common/application_code/main.c](#) ファイルを使用して、以下の操作を行います。

- 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();
...
}
```

- 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。

- 設定済みの WFINetworkParams_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。WFINetworkParams_t 構造の詳細については、「[使用例 \(p. 176\)](#)」および「[API リファレンス \(p. 176\)](#)」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"

INetworkParams_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(("%" : %d \n", xScanResults[i].cSSID, xScanResults[i].cRSSI));
        }
    }
}

```

```
        } else {
            configPRINTF(("Scan failed, status code: %d\n", (int)xWifiStatus));
        }

        vTaskDelay(200);
    }
```

移植

`aws_wifi.c` 実装は、`aws_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。

Amazon FreeRTOS デモ

Amazon FreeRTOS デモアプリケーションは、メインの Amazon FreeRTOS ディレクトリの下の `demos` フォルダにあります。Amazon 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 Amazon FreeRTOS 認定プラットフォーム用のフォルダもあります。[Amazon FreeRTOS コンソール](#)を使用する場合、選択したターゲットプラットフォームのみが `demos` の下にサブディレクトリを持っています。デモアプリケーションを自分で試す前に、[Amazon FreeRTOS の使用開始 \(p. 59\)](#) を使用して Hello World MQTT デモの設定と実行を案内することをお勧めします。

Amazon FreeRTOS デモを実行する

以下のトピックでは、Amazon FreeRTOS デモのセットアップと実行について説明します。

- Bluetooth Low Energy デモアプリケーション (p. 183)
- Microchip Curiosity PIC32MZEF 用のデモブートローダー (p. 194)
- AWS IoT Greengrass Discovery デモアプリケーション (p. 179)
- 無線による更新デモアプリケーション (p. 199)
- Secure Sockets Echo Client デモ (p. 202)
- AWS IoT Device Shadow デモアプリケーション (p. 181)

`AmazonFreeRTOS\demos\common\demo_runner\aws_demo_runner.c` にある、関数 `DEMO_RUNNER_RunDemos()` には、各例を呼び出すコードが含まれています。デフォルトでは、`DEMO_RUNNER_RunDemos()` は、Hello World MQTT デモを開始する `vStartMQTTEchoDemo()` 関数のみを呼び出します。Amazon 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 Amazon 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はコメントアウトされるか省略されます。

Note

例のすべての組み合わせが連携するわけではないことに注意してください。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 `demos/<vendor>/<platform>/common/config_files` にあります。

AWS IoT Greengrass Discovery デモアプリケーション

Amazon FreeRTOS の AWS IoT Greengrass Discovery デモを実行する前に、AWS、AWS IoT Greengrass、および AWS IoT を設定する必要があります。AWS を設定するには、[AWS アカウントとア](#)

「[クセス許可の設定 \(p. 61\)](#)」の指示に従います。AWS IoT Greengrass をセットアップするには、Greengrass グループを作成して Greengrass コアを追加する必要があります。AWS IoT Greengrass のセットアップの詳細については、「[AWS IoT Greengrass の開始方法](#)」を参照してください。

AWS と AWS IoT Greengrass を設定した後、AWS IoT Greengrass のいくつかの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。

[Amazon FreeRTOS コンソール](#) で [クイック接続] ワークフローを使用して、ボードをすばやく AWS IoT に接続し、デモを実行できます。現在、以下のボードでは Amazon FreeRTOS 設定は利用できません。

- Cypress CYW943907AEVAL1F 開発キット
- Cypress CYW954907AEVAL1F 開発キット
- Espressif ESP-WROVER-KIT
- Espressif ESP32-DevKitC
- Nordic nRF52840-DK

ボードを AWS IoT に接続して、手動で Amazon FreeRTOS デモを設定することもできます。

1. AWS IoT で MCU ボードを登録 (p. 61)

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。

新規 AWS IoT Greengrass ポリシーを作成する方法

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": [  
        "*"  
    ]  
}
```

このポリシーによって、すべてのリソースに AWS IoT Greengrass アクセス許可が付与されます。

- [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

- 「[AWS IoT コンソール](#)」を参照します。
- 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。
- [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。
- [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。
- 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. Amazon FreeRTOS のダウンロード (p. 64)

Note

Amazon FreeRTOS を Amazon FreeRTOS コンソールからダウンロードしている場合は、[Connect to AWS IoT- **Platform** (AWS IoT- #####に接続する)] ではなく [Connect to AWS IoT Greengrass- **Platform** (AWS IoT Greengrass- #####に接続する)] を選択します。

3. Amazon FreeRTOS デモを設定する (p. 64)

Amazon FreeRTOS を GitHub からダウンロードした場合は、<**BASE FOLDER**>/demos/common/demo_runner/aws_demo_runner.c を開き、extern void vStartMQTTEchoDemo(void); と vStartMQTTEchoDemo(); をコメントアウトし、extern void vStartGreenGrassDiscoveryTask(void); と vStartGreenGrassDiscoveryTask(); をコメント解除する必要があります。

AWS IoT と AWS IoT Greengrass を設定し、Amazon FreeRTOS をダウンロードして設定した後、IDE で Greengrass デモをビルドして実行できます。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 \(p. 59\)](#) の手順に従います。

Greengrass デモは、一連のメッセージを Greengrass コア、および AWS IoT MQTT クライアントに発行します。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] を選択して、freertos/demos/ggd にサブスクリプションを追加します。

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
```

AWS IoT Device Shadow デモアプリケーション

Device Shadow の例は、プログラムによってデバイスシャドウの変化を更新して対応する方法を示しています。このシナリオのデバイスは、色を赤または緑に設定できる電球です。Device Shadow のサンプルアプリケーションは、AmazonFreeRTOS/demos/common/shadow ディレクトリにあります。この例では、3 つのタスクを作成します。

- prvShadowMainTask を呼び出す主なデモタスク。
- prvUpdateTask を呼び出すデバイス更新タスク。
- prvShadowUpdateTasks を呼び出す多数のシャドー更新タスク。

prvShadowMainTask は Device Shadow クライアントを初期化し、AWS IoT への MQTT 接続を開始します。次に、デバイス更新タスクを作成します。最後に、シャドウ更新タスクを作成して終了します。AmazonFreeRTOS/demos/common/shadow/aws_shadow_lightbulb_on_off.c で定義されている democonfigSHADOW_DEMO_NUM_TASKS 定数が、作成されたシャドウアップデートタスクの数を制御します。

prvShadowUpdateTasks は、最初の Thing Shadow ドキュメントを生成し、Device Shadow をドキュメントで更新します。その後、電球の色を赤色から緑色、緑色から赤色に変えることを要求して、Thing Shadow の desired 状態を定期的に更新する無限ループに入ります。

prvUpdateTask は、Device Shadow の desired 状態の変化に応答します。desired 状態が変更されると、このタスクは Device Shadow の reported 状態を更新して、新しい desired 状態が反映されるようにします。

1. デバイス証明書に、次のポリシーを追加します。

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "iot:Connect",  
            "Resource": "arn:aws:iot:us-west-2:123456789012:client/<yourClientId>"  
        },  
        {  
            "Effect": "Allow",  
            "Action": "iot:Subscribe",  
            "Resource": "arn:aws:iot:us-west-2:123456789012:topicfilter/$aws/things/  
thingName/shadow/*"  
        },  
        {  
            "Effect": "Allow",  
            "Action": "iot:Receive",  
            "Resource":  
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/  
thingName/shadow/*"  
        },  
        {  
            "Effect": "Allow",  
            "Action": "iot:Publish",  
            "Resource":  
                "arn:aws:iot:us-west-2:123456789012:topic/$aws/things/  
thingName/shadow/*"  
        }  
    ]  
}
```

2. aws_demo_runner.c の vStartShadowDemoTasks の宣言と呼び出しをコメント解除します。この関数は、prvShadowMainTask 関数を実行するタスクを作成します。

AWS IoT コンソールを使用してデバイスのシャドウを表示し、desired 状態と reported 状態が定期的に変更されていることを確認します。

1. AWS IoT コンソールの左側のナビゲーションペインで、[管理] を選択します。
2. [管理] で [モノ] を選択し、次にシャドウを表示したいモノを選択します。
3. [モノの詳細] ページで、左のナビゲーションペインから [シャドウ] を選択し、Thing Shadow を表示します。

デバイスとシャドウの操作方法の詳細については、「[Device Shadow のデータフロー](#)」を参照してください。

Bluetooth Low Energy デモアプリケーション

新しい Bluetooth Low Energy (BLE) ライブラリは、Amazon FreeRTOS パブリックベータリリースであり、変更される可能性があります。

概要

Amazon FreeRTOS BLE には 3 つのデモアプリケーションがあります。

MQTT over BLE (p. 189) デモ

このアプリケーションは、MQTT over BLE サービスの使用方法を示します。

Wi-Fi プロビジョニング (p. 190) デモ

このアプリケーションは、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 (p. 193) デモ

このアプリケーションは、Amazon FreeRTOS BLE 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。

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が必要です。また、[Amazon FreeRTOS Bluetooth デバイス用の iOS SDK \(p. 158\)](#) または [Amazon FreeRTOS Bluetooth デバイス用の Android SDK \(p. 157\)](#) も必要です。

Amazon FreeRTOS BLE 用の AWS IoT および Amazon Cognito のセットアップ

MQTT でデバイスを AWS IoT に接続するには、AWS IoT および Amazon Cognito 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

1. <https://aws.amazon.com> で AWS アカウントをセットアップします。
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]、[单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。モバイル SDK では Amazon Cognito を使用してデバイス認証を行っているため、BLE を使用するデモのためにデバイス認証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。AWS IoT REST API エンドポイントを書きとめておきます。

セットアップの詳細については、「[AWS IoT の使用開始](#)」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、[ユーザープールの管理] を選択します。
2. [Create a user pool] 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 ID プールを作成するには

1. Amazon Cognito コンソールを開き、[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. [Create Pool] 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。[許可] を選択し、認証された ID および認証されていない ID が Amazon Cognito にアクセスするための IAM ロールを作成します。
6. [ID プールの編集] を選択します。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](#)」を参照してください。

IAM ポリシーを作成して、認証された 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 [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 (INLINE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": "  
                arn:aws:iot:  
                    <region>:  
                    <account>/cert/  
                    <cert-id>  
            "  
        }  
    ]  
}
```

```
        "Resource": [
            "*"
        ]
    }
}
```

- [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 および Amazon Cognito の情報を手元に用意しておきます。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、エンドポイントおよび ID が必要です。

BLE 用 Amazon FreeRTOS 環境をセットアップする

環境をセットアップするには、[Amazon FreeRTOS Bluetooth Low Energy Library \(ベータ\) \(p. 148\)](#) を使用してマイクロコントローラに Amazon FreeRTOS をダウンロードし、Amazon FreeRTOS Bluetooth デバイス用モバイル SDK を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。

Amazon FreeRTOS BLE でマイクロコントローラの環境をセットアップするには

- Amazon FreeRTOS [GitHub](#) リポジトリの `feature/ble-beta` ブランチをダウンロードします。
- マイクロコントローラに Amazon FreeRTOS をセットアップします。

Amazon FreeRTOS 認定を受けたマイクロコントローラで Amazon FreeRTOS の使用を開始する方法については、「[Amazon FreeRTOS の開始方法](#)」で該当するボードのガイドを参照してください。

Note

デモは、Amazon FreeRTOS と移植された Amazon FreeRTOS BLE ライブラリを搭載した BLE 対応マイクロコントローラで実行できます。現在、Amazon FreeRTOS [MQTT over BLE \(p. 189\)](#) デモプロジェクトは、以下の BLE 対応デバイスに完全に移植されます。

- [STMicroelectronics STM32L4 ディスカバリキット IoT ノード](#) と STBTLE-1S BLE モジュール
- [Espressif ESP32-DevKitC](#) と [ESP-WROVER-KIT](#)
- [Nordic nRF52840-DK](#)

共通コンポーネント

Amazon 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
- BLE 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\common\network_manager\aws_iot_network_manager.c` の Amazon FreeRTOS ディレクトリにあります。Network Manager が Wi-Fi と BLE の両方にに対応している場合、デフォルトでは BLE を使ってデモが開始されます。BLE 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、あるネットワーク接続タイプを有効にするには、そのネットワーク接続タイプを `demos\vendor\board\common\config_files\aws_iot_network_config.h` の

configENABLED_NETWORKS パラメータに追加します。たとえば、BLE 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、lib\include\aws_iot_network.h の #define AWSIOT_NETWORK_TYPE から始まる行を参照してください。

Amazon FreeRTOS BLE Mobile SDK デモアプリケーション

Amazon FreeRTOS BLE Mobile SDK デモアプリケーションは、amazon-freertos-ble-android-sdk/app の [Amazon FreeRTOS Bluetooth デバイス用の Android SDK](#) および amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo の [Amazon FreeRTOS Bluetooth デバイス用の iOS SDK](#) 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。

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. [Amazon FreeRTOS Bluetooth デバイス用の iOS SDK \(p. 158\)](#) 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.xcodeproj 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. 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 ポリシーの名前。
 - mqttCustomTopic: 発行する MQTT トピック。

Note

AWS リージョンと AWS IoT ポリシーがあるように AWS と AWS IoT を設定する必要があります。

5. オープン amazon-freertos-ble-ios-sdk/Example/AmazonFreeRTOSDemo/AmazonFreeRTOSDemo/Support/awsconfiguration.json.

CognitoIdentity で、次の変数を再定義します。

- PoolId: Amazon Cognito ID プールの 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. [Amazon FreeRTOS Bluetooth デバイス用の Android SDK \(p. 157\)](#) ガインストールされていることを確認します。
2. amazon-freertos-ble-android-sdk/app/src/main/java/com/amazon/aws/freertosandroid/AuthenticatorActivity.java を開き、次の変数を再定義します。
 - AWS_IOT_POLICY_NAME: AWS IoT ポリシーの名前。
 - AWS_IOT_REGION: AWS リージョン。
 - COGNITO_POOL_ID: Amazon Cognito ID プールの ID。
 - COGNITO_REGION: AWS リージョン。
3. amazon-freertos-ble-android-sdk/app/src/main/java/com/amazon/aws/freertosandroid/MainActivity.java を開き、次の変数を再定義します。
 - BLE_DEVICE_MAC_ADDR: デバイスの MAC アドレス。
 - BLE_DEVICE_NAME: デバイスの名前。
 - MTU: マイクロコントローラとモバイルデバイスの間の目的の MTU。
4. オープン amazon-freertos-ble-android-sdk/app/src/res/raw/awsconfiguration.json.
CognitoIdentity で、次の変数を再定義します。
 - PoolId: Amazon Cognito ID プールの 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。

BLE 経由でマイクロコントローラ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で BLE デモプロジェクトを実行します。
2. モバイルデバイスで BLE 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
```

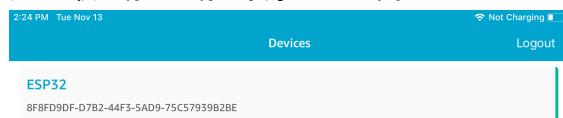
3. BLE Mobile SDK デモアプリの [デバイス] にマイクロコントローラが表示されていることを確認します。



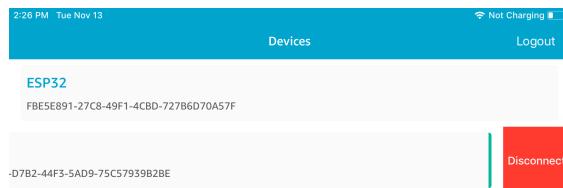
Note

このリストには、Amazon FreeRTOS を搭載したすべてのデバイスと、周囲にあるデバイス情報サービス (\lib\bluetooth_low_energy\services\device_information) が表示されます。

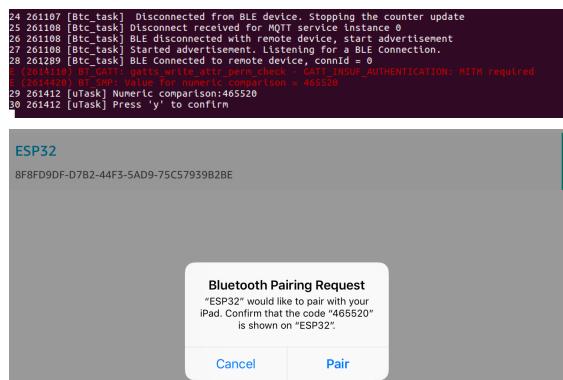
4. デバイスのリストからマイクロコントローラーを選択します。アプリケーションがボードとの接続を確立し、接続されたデバイスの横に緑色の線が表示されます。



マイクロコントローラーとの接続を切断するには、線を左へドラッグします。



5. マ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求められる場合があります。



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

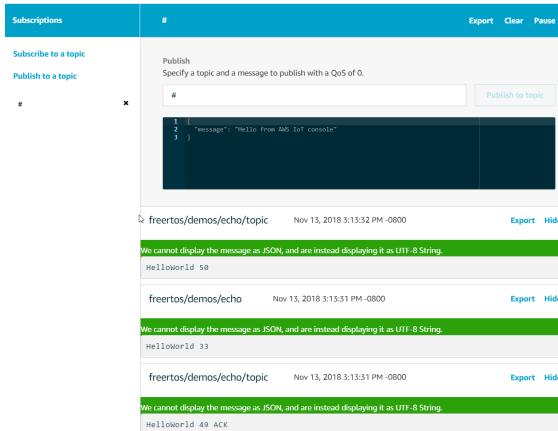
BLE Mobile SDK デモアプリケーションは、ユーザー認証に Amazon Cognito を使用します。Amazon Cognito ユーザーおよび ID プールを設定してあること、認証された ID に IAM ポリシーをアタッチしてあることを確認してください。

MQTT over BLE

MQTT over BLE デモでは、マイクロコントローラより AWS クラウドに MQTT プロキシ経由でメッセージが発行されます。

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、[Test (テスト)] 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「`freertos/demos/echo`」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。



MQTT デモは、BLE または Wi-Fi 接続経由実行できます。Network Manager (p. 185) の設定によって、使用される接続タイプが決まります。

BLE を使用してマイクロコントローラ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 BLE 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。

Wi-Fi を使用する場合、デモは、`demos/common/mqtt/aws_hello_world.c` にある MQTT Hello World デモプロジェクトと同じです。このデモは、「Amazon FreeRTOS の開始方法」のほとんどのデモプロジェクトで使用されます。

デモを有効化するには

デバイスの入門ガイドの指示に従ってすでに BLE デモを有効化している場合、これらの手順は省略できます。

1. Wi-Fi プロビジョニングサービスを有効にします。`demos\vendor\board\common\config_files\aws_ble_config.h` を開き、`#define bleconfigENABLE_WIFI_PROVISIONING` を 1 に設定します。

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。

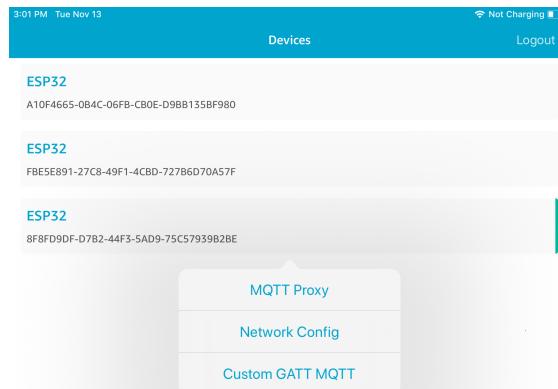
2. `demos\common\demo_runner\aws_demo_runner.c` を開き、デモの宣言の中で `extern void vStartMQTTBLEEchoDemo(void);` のコメントを解除します。`DEMO_RUNNER_RunDemos` 定義で、`vStartMQTTBLEEchoDemo()` のコメントを解除します。

デモを実行するには

Network Manager が Wi-Fi 接続のみに合わせて設定されている場合は、ボードでデモプロジェクトをビルドし、実行します。

Network Manager が BLE に合わせて設定されている場合は、次の操作を行います。

- マイクロコントローラーでデモプロジェクトをビルドし、実行します。
- ボードとモバイルデバイスが、[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#) を使ってペアリングされていることを確認します。
- 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。



- [Enable MQTT proxy (MQTT プロキシを有効化する)] をタッチします。スライダーが緑色に変わります。



MQTT プロキシを有効になると、`freertos/demos/echo` 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。

```
3 1993 [pthread] Received: HelloWorld 3
4 1994 [pthread] [INFO] [MQTT][1994] MQTT PUBLISH operation queued.
5 1994 [pthread] [INFO] [MQTT][1994] Waiting for operation PUBLISH to complete.
6 2002 [pthread] [INFO] [MQTT][2002] MQTT operation PUBLISH complete with result SUCCESS.
7 2002 [pthread] Sent: HelloWorld 3 ACK
8 2007 [pthread] [INFO] [MQTT][2007] MQTT PUBLISH operation queued.
9 2007 [pthread] [INFO] [MQTT][2007] Waiting for operation PUBLISH to complete.
10 2009 [pthread] [INFO] [MQTT][2009] MQTT operation PUBLISH complete with result SUCCESS.
11 2095 [pthread] Sent: HelloWorld 4
12 2101 [pthread] Received: HelloWorld 4
13 2101 [pthread] [INFO] [MQTT][2102] MQTT PUBLISH operation queued.
14 2102 [pthread] [INFO] [MQTT][2102] Waiting for operation PUBLISH to complete.
15 2110 [pthread] [INFO] [MQTT][2110] MQTT operation PUBLISH complete with result SUCCESS.
16 2110 [pthread] Sent: HelloWorld 4 ACK
```

Wi-Fi プロビジョニング

Wi-Fi プロビジョニングは、モバイルデバイスから Wi-Fi ネットワーク認証情報を BLE 経由で BLE 上のマイクロコントローラーに安全に送信する Amazon FreeRTOS BLE 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、`lib/bluetooth_low_energy/services/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。
デモモバイルアプリケーションの Android バージョンは、現在、Wi-Fi プロビジョニングをサポートしていません。

デモを有効化するには

- Wi-Fi プロビジョニングサービスを有効にします。`demos/vendor/board/common/config_files/aws_ble_config.h` を開き、`#define bleconfigENABLE_WIFI_PROVISIONING` を 1 に設定します。

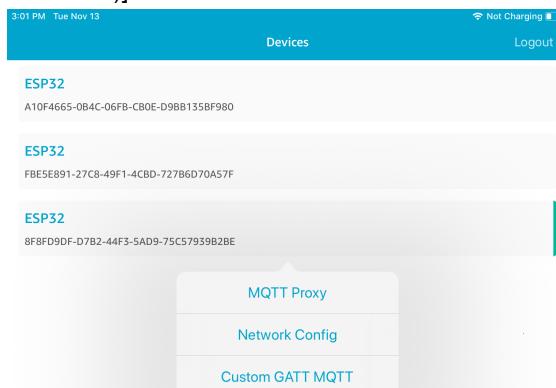
Note

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。

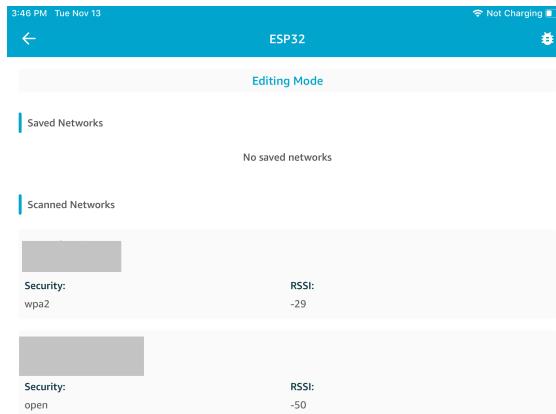
2. Wi-Fi と BLE の両方を有効にするよう [Network Manager \(p. 185\)](#) を設定します。

デモを実行するには

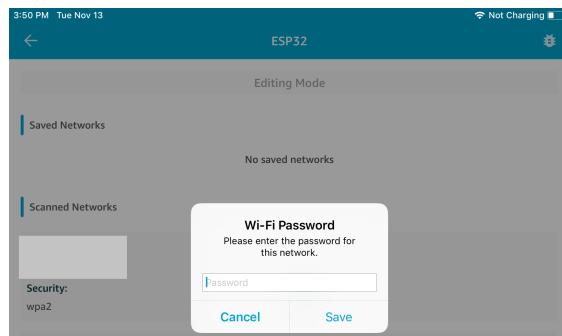
1. マイクロコントローラーでデモプロジェクトをビルドし、実行します。
2. マイクロコントローラーとモバイルデバイスが、[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#) を使ってペアリングされていることを確認します。
3. デモモバイルアプリの [Devices (デバイス)] リストからマイクロコントローラーを選択し、次に [Network Config (ネットワーク構成)] を選択して開きます。



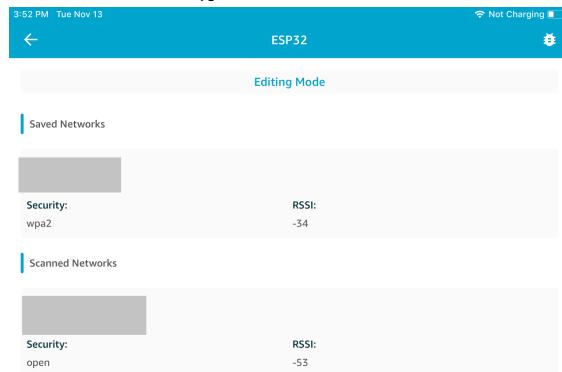
4. ボードの [Network Config (ネットワーク構成)] を選択した後、マイクロコントローラーが近隣にあるネットワークのリストをモバイルデバイスに送信します。[Scanned Networks (スキャンされたネットワーク)] のリストに利用可能な Wi-Fi ネットワークが表示されます。



[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。

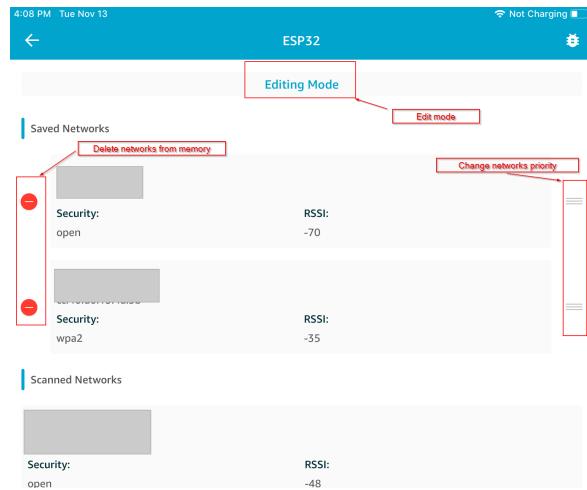


マイクロコントローラーがネットワークに接続し、ネットワークを保存します。ネットワークが [Saved Networks (保存されたネットワーク)] に表示されます。



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にあるデモの汎用属性 (GATT) サーバーアプリケーションが単純なカウンター値を [Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#) に送信します。

BLE 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. BLE GATT デモを有効化します。demos/[vendor/board/common/config_files/aws_ble_config.h](#) で、`#define bleconfigENABLE_GATT_DEMO (1)` を define ステートメントのリストに追加します。

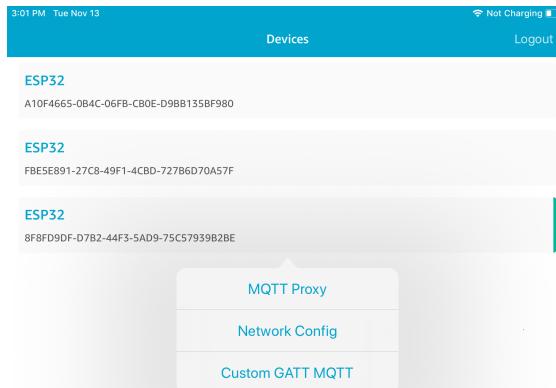
Note

BLE GATT デモはデフォルトで無効になっています。

2. demos\common\demo_runner\aws_demo_runner.c を開き、デモの宣言の中で `extern void vStartMQTTBLEEEchoDemo(void);` のコメントを解除します。DEMO_RUNNER_RunDemos 定義で、`vStartMQTTBLEEEchoDemo();` のコメントを解除します。

デモを実行するには

1. マイクロコントローラでデモプロジェクトをビルドし、実行します。
2. ボードとモバイルデバイスが、[Amazon FreeRTOS BLE Mobile SDK デモアプリケーション \(p. 186\)](#) 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. [Enable MQTT proxy (MQTT プロキシを有効化する)] をタッチします。スライダーが緑色に変わります。



5. [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。
6. [Start Counter (カウンターの開始)] を選択して freertos/demos/echo 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、`freertos/demos/echo` 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

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、Amazon 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。

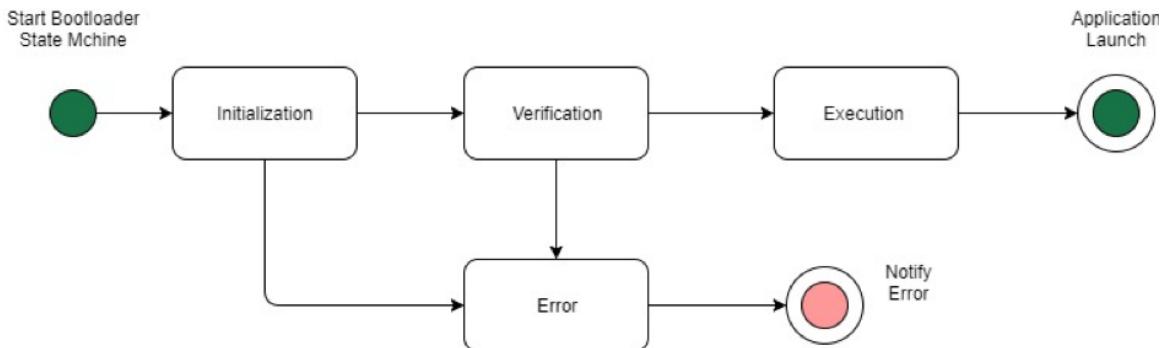
ファームウェア検証は、無線で受信した新しいファームウェアの信頼性と誠実性を検証することが含まれます。ブートローダーは、起動する前にアプリケーションの暗号署名を検証します。このデモでは、SHA256 上の楕円曲線のデジタル署名アルゴリズム (ECDSA) を使用しています。提供されているユーティリティを使用して、デバイス上でフラッシュできる署名付きアプリケーションを生成することができます。

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。

ブートローダーのステート

ブートローダープロセスは、次のステートマシンによって記述されます。



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。

ブートローダーのステート	説明
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 `Execution` の両方が `Execute Default` 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

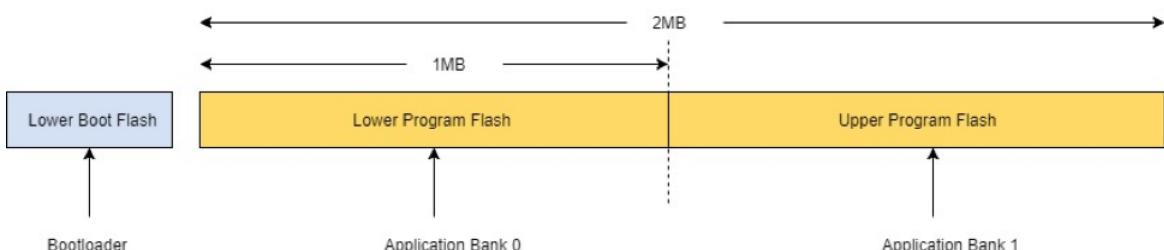
デフォルトイメージを起動する設定オプションが有効になっている場合、ブートローダーはデフォルトの実行アドレスからアプリケーションを起動します。デバッグ中を除いて、このオプションを無効にする必要があります。

ブートローダーのエラー状態

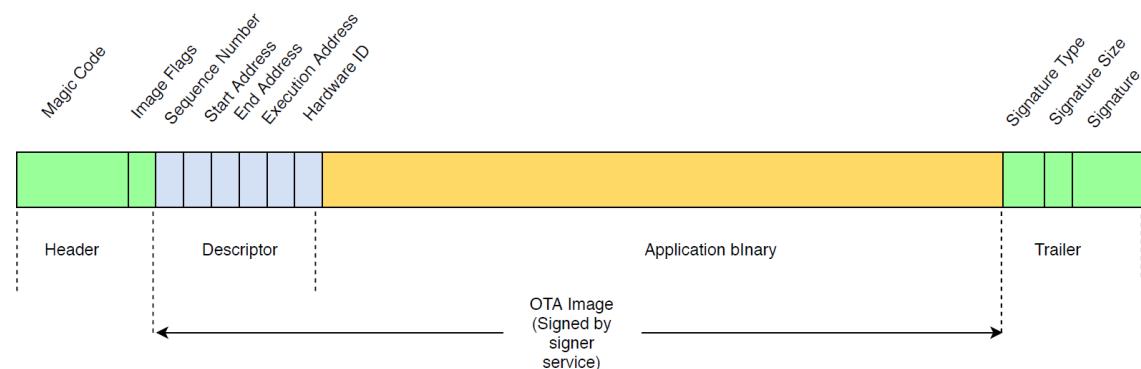
ブートローダーがエラー状態で、デバイスに有効なイメージがありません。ブートローダーはユーザーに通知する必要があります。デフォルトの実装では、ログメッセージがコンソールに送信され、ボード上の LED が永続的にしばらく点滅します。

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主要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

デバイスに存在する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。

ステータス	値	説明
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。このビルト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
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
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、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。aws_boot_config.h は /demos/microchip/curiosity_pic32_bl/config_files/ にあります。

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。

ブートローダーの構築

デモブートローダーは、Amazon FreeRTOS ソースコードリポジトリの `demos\microchip\curiosity_pic32mzef\mplab` の下にある `aws_demos` 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、`/demos/common/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、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` なしで構築し、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。

無線による更新デモアプリケーション

Amazon FreeRTOS には、OTA ライブラリの使用方法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、`demos\common\ota` サブディレクトリに配置されています。

OTA 更新を作成する前に、[Amazon FreeRTOS 無線による更新 \(p. 7\)](#) を読み、そこにリストされているすべての前提条件を満たしてください。

OTA デモアプリケーション

1. FreeRTOS ネットワークスタックおよび MQTT バッファーポールを初期化します。`(main.c` を参照してください。)
2. OTA ライブラリを使用するタスクを作成します。`(aws_ota_update_demo.c` の `vOTAUpdateDemoTask` を参照してください。)
3. `MQTT_AGENT_Create` を使用して MQTT クライアントを作成します。
4. `MQTT_AGENT_Connect` を使用して AWS IoT MQTT ブローカーに接続します。
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。

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。

```
313 267848 [OTA] [OTA] Queued: 1 Processed: 1 Dropped: 0
314 268733 [OTA Task] [OTA] Set job doc parameter [ jobId:
fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
315 268734 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
316 268734 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
317 268734 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
318 268735 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
319 268735 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
```

```
320 268735 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
321 268737 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa:
Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
322 268737 [OTA Task] [OTA] Job was accepted. Attempting to start transfer.
323 268737 [OTA Task] Sending command to MQTT task.
324 268737 [MQTT] Received message 50000 from queue.
325 268848 [OTA] [OTA] Queued: 2 Processed: 1 Dropped: 0
326 269039 [MQTT] MQTT Subscribe was accepted. Subscribed.
327 269039 [MQTT] Notifying task.
328 269040 [OTA Task] Command sent to MQTT task passed.
329 269041 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/streams/327

330 269848 [OTA] [OTA] Queued: 2 Processed: 1 Dropped: 0
... // Output removed for brevity
346 284909 [OTA Task] [OTA] file token: 74594452
.. // Output removed for brevity
363 301327 [OTA Task] [OTA] file ready for access.
364 301327 [OTA Task] [OTA] Returned buffer to MQTT Client.
365 301328 [OTA Task] Sending command to MQTT task.
366 301328 [MQTT] Received message 60000 from queue.
367 301328 [MQTT] Notifying task.
368 301329 [OTA Task] Command sent to MQTT task passed.
369 301329 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/
streams/327/get
370 301632 [OTA Task] [OTA] Received file block 0, size 1024
371 301647 [OTA Task] [OTA] Remaining: 127
... // Output removed for brevity
508 304622 [OTA Task] Sending command to MQTT task.
509 304622 [MQTT] Received message 70000 from queue.
510 304622 [MQTT] Notifying task.
511 304623 [OTA Task] Command sent to MQTT task passed.
512 304623 [OTA Task] [OTA] Published file request to $aws/bin/things/TI-LaunchPad/
streams/327/get
513 304860 [OTA] [OTA] Queued: 47 Processed: 47 Dropped: 83
514 304926 [OTA Task] [OTA] Received file block 4, size 1024
515 304941 [OTA Task] [OTA] Remaining: 82
... // Output removed for brevity
797 315047 [MQTT] MQTT Publish was successful.
798 315048 [MQTT] Notifying task.
799 315048 [OTA Task] Command sent to MQTT task passed.
800 315049 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/
jobs/fe18c7ec_8c31_4438_b0b9_ad55acd9561801 315049 [OTA Task] Sending command to MQTT task.
802 315049 [MQTT] Received message d0000 from queue.
803 315150 [MQTT] MQTT Unsubscribe was successful.
804 315150 [MQTT] Notifying task.
805 315151 [OTA Task] Command sent to MQTT task passed.
806 315152 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

807 315172 [OTA Task] Sending command to MQTT task.
808 315172 [MQTT] Received message e0000 from queue.
809 315273 [MQTT] MQTT Unsubscribe was successful.
810 315273 [MQTT] Notifying task.
811 315274 [OTA Task] Command sent to MQTT task passed.
812 315274 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPad/streams/327

813 315275 [OTA Task] [OTA] Resetting MCU to activate new image.
0 0 [Tmr Svc] Starting Wi-Fi Module ...
1 0 [Tmr Svc] Simple Link task created

Device came up in Station mode

2 137 [Tmr Svc] Wi-Fi module initialized.
3 137 [Tmr Svc] Starting key provisioning...
4 137 [Tmr Svc] Write root certificate...
5 243 [Tmr Svc] Write device private key...
6 339 [Tmr Svc] Write device certificate...
```

```
7 436 [Tmr Svc] Key provisioning done...
Device disconnected from the AP on an ERROR...!!!

[WLAN EVENT] STA Connected to the AP: Guest , BSSID: 44:48:c1:ba:b2:c3

[NETAPP EVENT] IP acquired by the device

Device has connected to Guest

Device IP Address is 192.168.3.72

8 1443 [Tmr Svc] Wi-Fi connected to AP Guest.
9 1444 [Tmr Svc] IP Address acquired 192.168.3.72
10 1444 [OTA] OTA demo version 0.9.1
11 1445 [OTA] Creating MQTT Client...
12 1445 [OTA] Connecting to broker...
13 1445 [OTA] Sending command to MQTT task.
14 1445 [MQTT] Received message 10000 from queue.
15 2910 [MQTT] MQTT Connect was accepted. Connection established.
16 2910 [MQTT] Notifying task.
17 2911 [OTA] Command sent to MQTT task passed.
18 2912 [OTA] Connected to broker.
19 2913 [OTA Task] Sending command to MQTT task.
20 2913 [MQTT] Received message 20000 from queue.
21 3014 [MQTT] MQTT Subscribe was accepted. Subscribed.
22 3014 [MQTT] Notifying task.
23 3015 [OTA Task] Command sent to MQTT task passed.
24 3015 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/$next/get/
accepted

25 3028 [OTA Task] Sending command to MQTT task.
26 3028 [MQTT] Received message 30000 from queue.
27 3129 [MQTT] MQTT Subscribe was accepted. Subscribed.
28 3129 [MQTT] Notifying task.
29 3130 [OTA Task] Command sent to MQTT task passed.
30 3138 [OTA Task] [OTA] Subscribed to topic: $aws/things/TI-LaunchPad/jobs/notify-next

31 3138 [OTA Task] [OTA] Check For Update #0
32 3138 [OTA Task] Sending command to MQTT task.
33 3138 [MQTT] Received message 40000 from queue.
34 3241 [MQTT] MQTT Publish was successful.
35 3241 [MQTT] Notifying task.
36 3243 [OTA Task] Command sent to MQTT task passed.
37 3245 [OTA Task] [OTA] Set job doc parameter [ clientToken: 0:TI-LaunchPad ]
38 3245 [OTA Task] [OTA] Set job doc parameter [ jobId:
fe18c7ec_8c31_4438_b0b9_ad55acd95610 ]
39 3245 [OTA Task] [OTA] Identified job doc parameter [ self_test ]
40 3246 [OTA Task] [OTA] Set job doc parameter [ updatedBy: 589827 ]
41 3246 [OTA Task] [OTA] Set job doc parameter [ streamname: 327 ]
42 3246 [OTA Task] [OTA] Set job doc parameter [ filepath: /sys/mcuflashimg.bin ]
43 3247 [OTA Task] [OTA] Set job doc parameter [ filesize: 130388 ]
44 3247 [OTA Task] [OTA] Set job doc parameter [ fileid: 126 ]
45 3247 [OTA Task] [OTA] Set job doc parameter [ attr: 0 ]
46 3247 [OTA Task] [OTA] Set job doc parameter [ certfile: tisigner.crt.der ]
47 3249 [OTA Task] [OTA] Set job doc parameter [ sig-sha1-rsa:
Q56qxHRq3Lxv6KkorvilVs4AyGJbWsJd ]
48 3249 [OTA Task] [OTA] Job is ready for self test.
49 3250 [OTA Task] Sending command to MQTT task.
51 3351 [MQTT] MQTT Publish was successful.
52 3352 [MQTT] Notifying task.
53 3352 [OTA Task] Command sent to MQTT task passed.
54 3353 [OTA Task] [OTA] Published 'IN_PROGRESS' status to $aws/things/TI-LaunchPad/jobs/
fe18c7ec_8c31_4438_b0b9_ad55acd95610/u55 3353 [OTA Task] Sending command to MQTT task.
56 3353 [MQTT] Received message 60000 from queue.
```

```
57 3455 [MQTT] MQTT Unsubscribe was successful.  
58 3455 [MQTT] Notifying task.  
59 3456 [OTA Task] Command sent to MQTT task passed.  
60 3456 [OTA Task] [OTA] Un-subscribed from topic: $aws/things/TI-LaunchPadstreams/327  
  
61 3456 [OTA Task] [OTA] Accepted final image. Commit.  
62 3578 [OTA Task] Sending command to MQTT task.  
63 3578 [MQTT] Received message 70000 from queue.  
64 3779 [MQTT] MQTT Publish was successful.  
65 3780 [MQTT] Notifying task.  
66 3780 [OTA Task] Command sent to MQTT task passed.  
67 3781 [OTA Task] [OTA] Published 'SUCCEEDED' status to $aws/things/TI-LaunchPad/jobs/  
fe18c7ec_8c31_4438_b0b9_ad55acd95610/upd68 3781 [OTA Task] [OTA] Returned buffer to MQTT  
Client.  
69 4251 [OTA] [OTA] Queued: 1 Processed: 1 Dropped: 0  
70 4381 [OTA Task] [OTA] Missing job parameter: execution  
71 4382 [OTA Task] [OTA] Missing job parameter: jobId  
72 4382 [OTA Task] [OTA] Missing job parameter: jobDocument  
73 4382 [OTA Task] [OTA] Missing job parameter: ts_ota  
74 4382 [OTA Task] [OTA] Missing job parameter: files  
75 4382 [OTA Task] [OTA] Missing job parameter: streamname  
76 4382 [OTA Task] [OTA] Missing job parameter: certfile  
77 4382 [OTA Task] [OTA] Missing job parameter: filepath  
78 4383 [OTA Task] [OTA] Missing job parameter: filesize  
79 4383 [OTA Task] [OTA] Missing job parameter: sig-sha1-rsa  
80 4383 [OTA Task] [OTA] Missing job parameter: fileid  
81 4383 [OTA Task] [OTA] Missing job parameter: attr  
82 4383 [OTA Task] [OTA] Returned buffer to MQTT Client.  
83 5251 [OTA] [OTA] Queued: 2 Processed: 2 Dropped: 0
```

Secure Sockets Echo Client デモ

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、[/common/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 Amazon FreeRTOS をダウンロードであること、Amazon FreeRTOS デモプロジェクトをビルドして実行してあることを確認します。Amazon FreeRTOS は [GitHub](#) からダウンロードします。Amazon FreeRTOS の認定を受けたボードのセットアップ方法については、「[Amazon FreeRTOS の開始方法](#)」を参照してください。

デモを実行するには

Note

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. TLS Echo Server をセットアップするには、[Amazon FreeRTOS 認定開発者ガイド](#)の「TLS Server Setup (TLS サーバー設定)」セクションの手順に従います。

ステップ 6 を完了すると、TLS Echo Server がポート 9000 で実行され、リッスンします。ステップ 7 ~ 9 を行う必要はありません。

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 行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 内の cTlsECHO_SERVER_CERTIFICATE_PEM にコピーします。
8. demos/common/demo_runneraws_demo_runner.c で、デモ関数を vStartTCPEchoClientTasks_SingleTasks() に切り替えます。

```
//extern void vStartMQTTEchoDemo( void );
extern void *vStartTCPEchoClientTasks_SingleTasks*( void );

/**
 * @brief Runs demos in the system.
 */
void DEMO_RUNNER_RunDemos( void )
{
    //vStartMQTTEchoDemo();
    vStartTCPEchoClientTasks_SingleTasks();
}
```

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。

デバイスの移植

この移植ガイドでは、Amazon FreeRTOS ソフトウェアパッケージを修正して Amazon FreeRTOS 認定されていないボード上で動作させる方法について説明します。Amazon FreeRTOS はボードやアプリケーションに必要なライブラリだけを選択できるように設計されています。MQTT、Shadow、Defender、および Greengrass ライブラリはほとんどのデバイスと互換性を持つように設計されているため、これらのライブラリの移植ガイドはありません。

Amazon FreeRTOS ライブラリを移植する前に、FreeRTOS カーネルをデバイスに移植する必要があります。FreeRTOS カーネル移植の詳細については、「[FreeRTOS Kernel Porting Guide](#)」を参照してください。

Amazon FreeRTOS の移植とテストの詳細については、「[Amazon FreeRTOS 認定開発者ガイド](#)」を参照してください。

トピック

- [Bootloader \(p. 204\)](#)
- [ログ記録 \(p. 204\)](#)
- [接続 \(p. 205\)](#)
- [セキュリティ \(p. 206\)](#)
- [Amazon FreeRTOS のカスタムライブラリを使用する \(p. 208\)](#)
- [OTA ポータブル抽象化レイヤー \(p. 208\)](#)

Bootloader

ブートローダーは、デュアルバンク対応である必要があり、イメージヘッダー内の CRC およびアプリケーションのバージョンをチェックするロジックが含まれます。ブートローダーは、CRC が有効であれば、ヘッダーのアプリケーションのバージョンに基づいて最新のイメージを起動します。CRC チェックが失敗した場合、ブートローダーは今後の再起動のための最適化としてヘッダーをフォーマットする必要があります。

OTA v1 エージェントは、暗号化署名検証を実行するため、v1 ブートローダーは暗号化コードにリンクせずに可能な限り小さくすることをお勧めします。互換性のあるブートローダーを用意する必要があります。

ログ記録

Amazon FreeRTOS では、configPRINTF 関数を呼び出して使用できるスレッドセーフなロギングタスクが提供されています。configPRINTF は printf のように動作するように設計されています。configPRINTF を移植するには、コミュニケーション周辺機器を初期化して configPRINT_STRING マクロを定義すると、入力文字列を受け取って任意の出力に表示されます。

ログ作成設定

ボードにログ記録を実装するには、configPRINT_STRING を定義する必要があります。現在の例では UART シリアルターミナルを使用していますが、その他のインターフェイスも使用できます。

```
#define configPRINT_STRING( x )
```

configLOGGING_MAX_MESSAGE_LENGTH を使用して、印刷する最大バイト数を設定します。この長さを超えるメッセージは切り捨てられます。

```
#define configLOGGING_MAX_MESSAGE_LENGTH
```

configLOGGING_INCLUDE_TIME_AND_TASK_NAME が 1 に設定されている場合、すべての印刷されたメッセージの前に追加のデバッグ情報(メッセージ番号、FreeRTOS ティックカウント、タスク名)が付加されます。

```
#define configLOGGING_INCLUDE_TIME_AND_TASK_NAME     1
```

vLoggingPrintf は FreeRTOS スレッドセーフ printf コールの名前です。AmazonFreeRTOS ロギングを使用するには、この値を変更する必要はありません。

```
#define configPRINTF( x )      vLoggingPrintf x
```

接続

最初に接続周辺機器を設定する必要があります。Wi-Fi、Bluetooth、イーサネット、またはその他の接続メディアを使用できます。現時点では、Wi-Fi 管理 API のみがボードポート用に定義されていますが、イーサネットを使用している場合、[FreeRTOS TCP/IP software](#) を使用して始めることをお勧めします。

Wi-Fi 管理

Wi-Fi 管理ライブラリでは、802.11 (a/b/n) プロトコルに従ったネットワーク接続がサポートされています。Wi-Fi をサポートしていないハードウェアの場合、このライブラリを移植する必要はありません。

移植する必要がある関数は、lib/wifi/portable/<vendor>/<platform>/aws_wifi.c ファイルに一覧表示されます。lib/include/aws_wifi.h に、パブリックインターフェイスごとの詳細な説明があります。

以下の関数を移植する必要があります。

```
WiFiReturnCode_t WIFI_On( void );
WiFiReturnCode_t WIFI_Off( void );
WiFiReturnCode_t WIFI_ConnectAP( const WiFiNetworkParams_t * const pxNetworkParams );
WiFiReturnCode_t WIFI_Disconnect( void );
WiFiReturnCode_t WIFI_Reset( void );
WiFiReturnCode_t WIFI_Scan( WiFiScanResult_t * pxBuffer, uint8_t uxNumNetworks );
```

ソケット

ソケットライブラリは、ボードとネットワーク内の別のノード間での TCP/IP ネットワーク通信をサポートします。ソケット API は、バーカレーソケットインターフェイスに基づいていますが、安全な通信オプションも含まれています。現時点では、クライアント API のみがサポートされています。TLS 機能を追加する前に、TCP/IP 機能を移植することをお勧めします。

MQTT、Shadow、および Greengrass 用のライブラリは、すべてソケットレイヤーに呼び出しを行います。移植が成功したソケットレイヤーは、ソケット上に構築されたプロトコルを動作可能にします。

バークレーソケット実装との主な相違点

セキュリティ

ソケットインターフェイスは、安全な通信に TLS を使用するように設定されている必要があります。SetSockOpt コマンドには、AmazonFreeRTOS の例で動作するように実装する必要があるいくつかの標準外オプションがあります。

```
SOCKETS_SO_REQUIRE_TLS  
SOCKETS_SO_SERVER_NAME_INDICATION  
SOCKETS_SO_TRUSTED_SERVER_CERTIFICATE
```

これらの標準外オプションの詳細については、「[セキュアソケットのドキュメント \(p. 168\)](#)」を参照してください。TLS と暗号化オペレーションの移植に関する詳細は、「[TLS \(p. 173\)](#)」および「[公開キー暗号化標準 #11 \(p. 166\)](#)」セクションを参照してください。

エラーコード

SOCKETS ライブラリは、グローバルエラーを設定するのではなく、API からエラーコードを返します。返されるすべてのエラーコードは、負の値である必要があります。

移植する必要があるパブリックインターフェイスは、lib/secure_sockets/portable/<vendor>/<platform>/aws_secure_sockets.c に一覧表示されます。

lib/include/aws_secure_sockets.h に、パブリックインターフェイスごとの詳細な説明があります。

mbed TLS に基づいた TLS を使用している場合は、プレーンテキストまたは暗号化されたバッファを送受信するために TLS レイヤーに登録されたネットワーク送受信機能を実装することでリファクタリングの労力を削減できます。

セキュリティ

Amazon FreeRTOS には、プラットフォームのセキュリティを提供するために連携して機能する 2 つのライブラリ、TLS と PKCS#11 があります。Amazon FreeRTOS は mbed TLS (サードパーティの TLS ライブラリ) 上に構築されたソフトウェアセキュリティソリューションを提供します。TLS API は、mbed TLS を使用してネットワークトラフィックを暗号化および認証します。PKCS#11 では、暗号化マテリアルを処理してソフトウェア暗号化オペレーションを、ハードウェアを最大限に使用する実装に置き換える標準インターフェイスを提供します。

TLS

mbed TLS ベースの実装を使用する場合は、PKCS#11 が実装されていれば aws_tls.c as-is を使用することができます。

このライブラリのパブリックインターフェイスと TLS インターフェイスごとの詳細な説明は、lib/include/aws_tls.h に一覧表示されています。TLS ライブラリの Amazon FreeRTOS 実装は、lib/tls/aws_tls.c にあります。独自の TLS サポートを使用する場合は、TLS パブリックインターフェイスを実装してソケットパブリックインターフェイスに接続するか、独自の TLS インターフェイスを使用して、ソケットライブラリを直接移植します。

mbedtls_hardware_poll 関数では、決定論的ランダムビットジェネレーターのランダム性を提供しています。セキュリティのためには、2 つのボードに同じランダム性を提供しないこと、およびボードがリセットされても、繰り返し同じランダム値を提供しない事が必要です。この関数の実装の例は、demos\<vendor>\<platform>\common\application_code\<vendor code>\aws_entropy_hardware_poll.c の mbed TLS を使用した移植に記載されています。

mbed TLS 以外の TLS ライブラリを使用する

別の TLS ライブラリを Amazon FreeRTOS に移植する場合は、互換性のある TLS 暗号スイートが実装されていることを確認します。詳細については、[AWS IoT と互換性のある暗号スイート](#)を参照してください。以下の暗号スイートは、AWS IoT Greengrass デバイスと互換性があります。

- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (非推奨)
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (非推奨)
- TLS_RSA_WITH_AES_128_CBC_SHA (非推奨)
- TLS_RSA_WITH_AES_256_CBC_SHA (非推奨)

SHA 1 での攻撃が原因で、Amazon FreeRTOS 接続には SHA 256 または SHA 384 を使用することをお勧めします。

PKCS #11

Amazon FreeRTOS は、暗号化オペレーションとキーストレージに PKCS#11 標準を実装しています。PKCS#11 用ヘッダーファイルは業界標準です。PKCS#11 を移植するには、不揮発性メモリ (NVM) との間で認証情報を読み書きする関数を実装する必要があります。

実装する必要がある関数は、lib/third_party/pkcs11/pkcs11f.h に一覧表示されています。パブリックインターフェイスの実装は、lib/pkcs11/portable/vendor/board/pkcs11.c にあります。

Amazon FreeRTOS で TLS クライアント認証をサポートするには、以下の関数が最小限必要です。

- C_GetFunctionList
- C_Initialize
- C_GetSlotList
- C_OpenSession
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_GetAttributeValue
- C_FindObjectsInit
- C_FindObjects
- C_FindObjectsFinal
- C_SignInit
- C_Sign
- C_CloseSession
- C_Finalize

一般的な移植ガイドについては、オープンスタンダードの [PKCS#11 Cryptographic Token Interface Base Specification](#) を参照してください。

共通の mbedTLS ベースの PKCS #11 レイヤーを使用して、ポート用に 4 つの追加の 非 PKCS #11 標準関数を実装する必要があります。

`PKCS11_PAL_FindObject`

PKCS #11 ラベルをオブジェクトハンドルに変換します。

`PKCS11_PAL_GetObjectValue`

ハンドル内のストレージのオブジェクトの値を取得します。この呼び出しほは、オブジェクト値データがコピーされるバッファを動的に割り当てます。使用するたびに `PKCS11_PAL_GetObjectValueCleanup` を呼び出して、動的に割り当てられたバッファを解放します。

`PKCS11_PAL_GetObjectValueCleanup`

`PKCS11_PAL_GetObjectValue` を呼び出した後、動的に割り当てられたバッファを消去します。

`PKCS11_PAL_SaveObject`

ローカルストレージにファイルを書き込みます。

Amazon FreeRTOS のカスタムライブラリを使用する

すべての Amazon FreeRTOS ライブラリを、カスタム開発ライブラリに置き換えることができます。すべてのカスタムライブラリは、置き換えられた Amazon FreeRTOS ライブラリの API に準拠している必要があります。

OTA ポータブル抽象化レイヤー

Amazon FreeRTOS では、OTA ライブラリがさまざまなハードウェアで役立つことを確認するために OTA ポータブル抽象化レイヤー (PAL) を定義しています。OTA PAL インターフェイスは下記のとおりです。

`prvAbort`

OTA 更新を中止します。

`prvCreateFileForRx`

受信したデータチャンクを保存する新しいファイルを作成します。

`prvCloseFile`

指定されたファイルを閉じます。これはファイルが安全であるとしてマークされている場合、ファイルを認証します。

`prvCheckFileSignature`

指定したファイルの署名を確認します。ビルトインの署名検証を実施しているデバイスファイルシステムの場合、冗長性備えている可能性があるため、no-op として実装します。

`prvWriteBlock`

指定されたファイルに、指定されたオフセットでデータブロックを書き込みます。正常または負のエラーコードで書き込まれたバイト数を返します。

`prvActivateNewImage`

新しいファームウェアイメージをアクティブ化します。一部のポートでは、この関数を返さない場合があります。

prvSetImageState

最後のファームウェアイメージ (またはバンドル) を受け入れるか拒否するために、プラットフォームで必要なすべてのことをしますか。お客様のプラットフォームでの動作を明確にするには、プラットフォームの実装を参照してください。

prvReadAndAssumeCertificate

指定された署名者証明書をファイルシステムから読み込み、発信者に返します。これは一部のプラットフォームでオプションです。

デバイスに資格を与える

AWS デバイス認定プログラムにより Amazon FreeRTOS の資格をデバイスに与えることができます。AWS デバイス認定プログラムの詳細については、[AWS パートナーネットワーク](#)をご覧ください。認定済みデバイスの一覧については、[AWS Partner Device Catalog](#) を参照してください。

Note

Amazon FreeRTOS 資格についてデバイスを検証するには、デバイスに Amazon FreeRTOS をポートする必要があります。ポートする方法については、[デバイスの移植 \(p. 204\)](#) を参照してください。

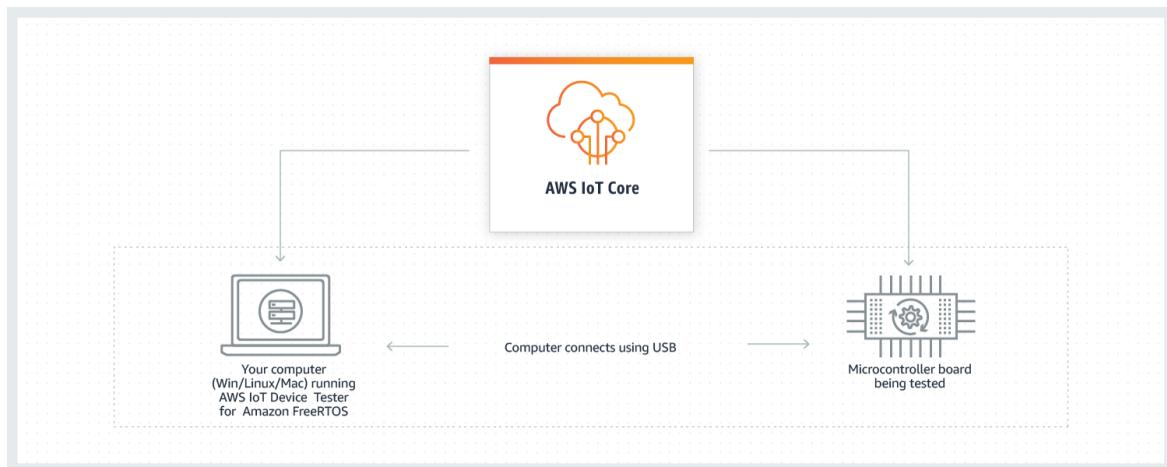
AWS IoT Device Tester for Amazon FreeRTOS を使用する

AWS IoT Device Tester (IDT) を使用すると、Amazon FreeRTOS オペレーティングシステムがデバイスでローカルに動作していて AWS IoT クラウドと通信できていることをテストできます。AWS IoT Device Tester は、Amazon FreeRTOS ライブラリのポーティングレイヤーがマイクロコントローラーボードデバイスドライバーとの組み合わせで正しく機能しているかを点検します。さらに、AWS IoT Core のエンドツーエンドのテストも実行します(たとえば、ボードが MQTT メッセージの送受信と処理を実行できるかどうかをテストします)。AWS IoT Device Tester for Amazon FreeRTOS は、[Amazon FreeRTOS GitHub リポジトリ](#)で発行されているテストケースを使用します。AWS IoT Device Tester は、Test Manager コマンドラインツールと一緒にテストケースで構成されます。

Test Manager 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、Mac、または Linux) 上で動作します。Test Manager 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。テストケースは、テストログを含み、テストに必要なリソースを設定します。

テストケースは、ボード上にフラッシュされたアプリケーションのバイナリイメージに含まれています。アプリケーションのバイナリイメージには、Amazon FreeRTOS、半導体ベンダーの移植 Amazon 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ます。テストケースは、埋め込みとして実行され、移植された Amazon 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します。

次の図は、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。



AWS IoT Device Tester for Amazon FreeRTOS バージョン

以下のタブでは、AWS IoT Device Tester for Amazon FreeRTOS のバージョンについて説明しています。ソフトウェアをダウンロードすることにより、[AWS IoT Device Tester ライセンス契約](#)に同意したと見なされます。Amazon FreeRTOS のバージョンごとにに対応する AWS IoT Device Tester のバージョンがあります。Amazon FreeRTOS が新しくリリースされるたびに、AWS IoT Device Tester の新しいバージョンがこのページに追加されます。AWS IoT Device Tester for Amazon FreeRTOS の詳細については、[AWS IoT Device Tester for Amazon FreeRTOS ユーザーガイド](#)を参照してください。

IDT for Amazon FreeRTOS v1.4

IDT for Amazon FreeRTOS v1.4.7 - 現行バージョン

- IDT v1.1 for Amazon FreeRTOS - [Linux](#) (MD5 fe843e96bca65b0dc4f7f1a54e8051ab)
- IDT v1.1 for Amazon FreeRTOS - [Mac](#) (MD5 feb337b6667a4764de725b6c05549083)
- IDT v1.1 for Amazon FreeRTOS - [Windows](#) (MD5 979c39f5ce0550079a02fe48feed0c36)

IDT for Amazon FreeRTOS v1.4.6

- IDT v1.0 for Amazon FreeRTOS - [Linux](#) (MD5 bb4290f556b7c828ec9621c7229b1d6d)
- IDT v1.0 for Amazon FreeRTOS - [Mac](#) (MD5 d5f62d6c1a253ef1fb0c70f2a87836fc)
- IDT v1.0 for Amazon FreeRTOS - [Windows](#) (MD5 9be81e92e7b1c5a85f8a4d433fc2bb43)

IDT for Amazon FreeRTOS v1.4.5

- IDT v1.0 for Amazon FreeRTOS - [Linux](#) (MD5 ee7b167e4e1cc02c256b5c3edc0a8abe)
- IDT v1.0 for Amazon FreeRTOS - [Mac](#) (MD5 791580fa714ef1a01c3c6bc32c2ff448)
- IDT v1.0 for Amazon FreeRTOS - [Windows](#) (MD5 ba744c23f6276bba928775bc024ec108)

IDT for Amazon FreeRTOS v1.4.4

- IDT v1.0 for Amazon FreeRTOS - [Linux](#) (MD5 64613157503fc55ce581310f819d6b3c)
- IDT v1.0 for Amazon FreeRTOS - [Mac](#) (MD5 b15840ab9dc14f076ce36cb9d111070f)
- IDT v1.0 for Amazon FreeRTOS - [Windows](#) (MD5 b84178c6cf98b376f50c360eb7a5208f)

IDT for Amazon FreeRTOS v1.4.3

- IDT v1.0 for Amazon FreeRTOS - [Linux](#) (MD5 49946aec8634d20d38c449f39ac58fef)
- IDT v1.0 for Amazon FreeRTOS - [Mac](#) (MD5 c8cc6c388a35482ea50f1693b40b5e72)
- IDT v1.0 for Amazon FreeRTOS - [Windows](#) MD5 a1ca9101729bc41ad29fc32458c8920d)

IDT for Amazon FreeRTOS v1.4.2

- IDT v1.0 for Amazon FreeRTOS - [Linux](#) (MD5 eb9cf1ae803d309f8d8b6698e40ae96)
- IDT v1.0 for Amazon FreeRTOS - [Mac](#) (MD5 10cb6c8aa2a54f7a0d26095f18753c2f)
- IDT v1.0 for Amazon FreeRTOS - [Windows](#) (MD5 57ac5703924e3a2e2f039e97baaca44f)

前提条件

このセクションでは、AWS IoT Device 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。

Amazon FreeRTOS をダウンロードする

テストしたい Amazon FreeRTOS のバージョンを [GitHub](#) からダウンロードします。Windows を使用している場合は、ファイルパスを短くするよう心がけてください。たとえば、長いファイルパスに対する Windows の制限を回避するため、`C:\Users\username\programs\projects\AmazonFreeRTOS\` ではなく `C:\AFreeRTOS` のクローンを作成します。

AWS IoT Device Tester for Amazon FreeRTOS をダウンロードする

Amazon FreeRTOS のどのバージョンにも、適格性確認テストを行うために、対応するバージョンの AWS IoT Device Tester が用意されています。適切なバージョンの AWS IoT Device Tester を [AWS IoT Device Tester for Amazon FreeRTOS バージョン \(p. 211\)](#) からダウンロードします。

AWS IoT Device Tester 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てください。Microsoft Windows では、パスの長さに制限があるため、AWS IoT Device Tester を C:\ や D:\ のようなルートディレクトリに抽出します。

AWS アカウントを作成して設定する

AWS アカウントをお持ちでない場合は、[AWS ウェブページ](#)の手順に従ってアカウントを作成してください。[AWS アカウントの作成] を選択し、プロンプトに従います。

AWS アカウント内で IAM ユーザーを作成する

AWS アカウントを作成すると、アカウント内のすべてのリソースへのアクセス権を持つルートユーザーが作成されます。日常的なタスクを行うための別のユーザーを作成しておくことをお勧めします。IAM ユーザーを作成するには、「[AWS アカウントでの IAM ユーザーの作成](#)」の手順に従います。ルートユーザーの詳細については、「[AWS アカウントのルートユーザー](#)」を参照してください。

IAM ポリシーを作成して AWS アカウントにアタッチする

IAM ポリシーでは、IAM ユーザーに AWS リソースへのアクセス権を付与します。

IAM ポリシーを作成するには

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、[Policies] を選択し、[Create Policy] を選択します。
3. [JSON] タブを選択し、[アクセス許可ポリシーテンプレート \(p. 230\)???](#) (p. 230)にあるポリシーテンプレートをコピーしてエディタウィンドウに貼り付けます。
4. [ポリシーの確認] を選択します。
5. [名前] に、ポリシーの名前を入力します。[説明] に説明を入力します(省略可能)。[ポリシーの作成] を選択します。

IAM ポリシーを作成したら、それを IAM ユーザーにアタッチする必要があります。

IAM ポリシーを IAM ユーザーにアタッチするには

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 [ユーザー] を選択します。IAM ユーザーを見つけて選択します。
3. [Add permissions (アクセス許可の追加)]、[Attach existing policies directly (既存のポリシーを直接アタッチする)] の順に選択します。IAM ポリシーを見つけて選択し、[Next: Review (次へ: レビュー)]、[Add Permissions (アクセス許可の追加)] の順に選択します。

AWS コマンドラインインターフェイス (CLI) をインストールする

何らかの操作に CLI が必要で、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](#)」の手順に従います。

マイクロコントローラー ボードの適格性確認を初めて行う

Amazon FreeRTOS インターフェイスを移植する際、AWS IoT Device Tester を使用してテストすることができます。ボードのデバイスドライバーのために Amazon 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 ボードの適格性確認テストを実行します。

ライブラリポートイングレイヤーを追加する

お使いの MCU アーキテクチャーと互換性がある Amazon FreeRTOS デバイスライブラリ用ライブラリポートイングレイヤー (TCP/IP、WiFi など) を追加するには、以下を行なう必要があります。

1. AWS IoT Device Tester テストを実行する前に configPRINT_STRING() メソッドを実装します。AWS IoT Device Tester が configPRINT_STRING() マクロを呼び出し、テスト結果を人間が読み取れる ASCII 文字列として出力します。
2. ドライバを移植して Amazon FreeRTOS ライブラリのインターフェイスを実装します。詳細については、「[Amazon FreeRTOS 認定開発者ガイド](#)」を参照してください。

AWS 認証情報を設定する

AWS 認証情報を `<devicetester_extract_location>/devicetester_afreertos_[win|mac|linux]/configs/config.json` で設定する必要があります。以下のいずれかの方法で認証情報を指定できます。

- 環境変数
- 認証情報ファイル

環境変数を使用した AWS 認証情報の設定

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。AWS IoT Device Tester 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。環境変数の設定方法は、実行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。

Windows で環境変数を設定するには

1. Windows 10 のデスクトップから Power User タスクメニューを開きます。このメニューを開くには、マウスカーソルを画面左下 ([スタート] メニューアイコン) に移動し、右クリックします。
2. Power User タスクメニューで [システム]、[システムの詳細設定] の順に選択します。

Note

Windows 10 では、必要に応じて [関連設定] までスクロールし、[システム情報] を選択します。[システム] で [システムの詳細設定] を選択します。

[システムのプロパティ] で [詳細設定] タブを選択し、[環境変数] ボタンを選択します。

3. [~ のユーザー環境変数<user-name>] で、[新規] を選択し、新しい環境変数を作成します。各環境変数の名前と値を入力します。

macOS、Linux、または UNIX で環境変数を設定するには

- 任意のテキストエディタで `~/.bash_profile` を開き、以下の行を追加します。

```
export AWS_ACCESS_KEY_ID="<your-aws-access-key-id>"  
export AWS_SECRET_ACCESS_KEY="<your-aws-secret-key>"
```

山かっこ (<>) 内の項目を AWS アクセスキー ID と AWS シークレットキーに置き換えます。

環境変数の設定後に、コマンドラインウィンドウまたはターミナルを閉じて再び開き、新しい値を有効にします。

環境変数を使用して AWS 認証情報を設定するには、`AWS_ACCESS_KEY_ID` と `AWS_SECRET_ACCESS_KEY` を設定します。`config.json` で、`method` の `environment` を設定します。

```
{  
  "awsRegion": "us-west-2",  
  "auth": {  
    "method": "environment"  
  }  
}
```

認証情報ファイルを使った AWS 認証情報の設定

認証情報を含む認証情報ファイルを作成します。AWS IoT Device Tester は、AWS CLI と同じ認証情報ファイルを使用します。詳細については、「」を参照してください。名前付きプロファイルも指定する必要があります。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。以下は、`config.json` ファイル内の認証情報ファイルを使用して AWS 認証情報を指定する方法を示す JSON スニペットの例です。

```
{  
  "awsRegion": "us-west-2",  
  "auth": {  
    "method": "file",  
    "credentials": {  
      "profile": "default"  
    }  
  }  
}
```

AWS リージョンの設定

AWS IoT Device Tester は、AWS アカウントにクラウドリソースを作成します。テストに使用する AWS リージョンを指定するには、`config.json` ファイルで `awsRegion` パラメータを設定します。デフォルトでは、AWS IoT Device Tester は、`us-west-2` リージョンを使用します。

AWS IoT Device Tester 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、同じ使用を持つデバイスで構成されます。Device Tester 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。適格性確認プロセスを加速させるために、AWS IoT Device Tester は、同じ仕様を持つデバイスを並行してテストすることができます。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 つのデバイスをテストするときの AWS IoT Device Tester の設定

configs フォルダの device.json ファイルを編集することでデバイスプールを定義します。次の例では、device.json ファイルを使って 1 つ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": "OTA",  
        "value": "Yes | No"  
      },  
      {  
        "name": "TCP/IP",  
        "value": "On-chip | Offloaded | No"  
      },  
      {  
        "name": "TLS",  
        "value": "On-chip | Offloaded | No"  
      }  
    ],  
    "devices": [  
      {  
        "id": "<device-id>",  
        "connectivity": {  
          "protocol": "uart",  
          "serialPort": "<serial-port>"  
        },  
        "identifiers": [  
          {  
            "name": "serialNo",  
            "value": "<serialNo-value>"  
          }  
        ]  
      }  
    ]  
  }]  
]
```

次のリストは、device.json ファイルで使用される属性を説明しています。

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カ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 にボードを出品する場合は、ここで指定する SKU と出品プロセスで使用する SKU が一致しなければなりません。

features

デバイスでサポートされている機能を含む配列。Device Tester は、この情報に基づいて実行する適格性確認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、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。
- **WIFI**: ボードが Wi-Fi 機能を備えているかを示します。
- **TLS**: ボードが TLS をサポートしているか、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。
- **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。

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。

複数のデバイスをテストするときの AWS IoT Device Tester の設定

複数のデバイスを追加するには、`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。`device.json` ファイルの構造と内容の詳細については、[1つのデバイスをテストするときの AWS IoT Device Tester の設定 \(p. 216\)](#) を参照してください。

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。

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ピールを作成します。

```
[{"id": "<pool-id>",  
 "sku": "<sku>",  
 "features": [  
   {  
     "name": "WIFI",  
     "value": "Yes | No"  
   },  
   {  
     "name": "OTA",  
     "value": "Yes | No"  
   },  
   {  
     "name": "TCP/IP",  
     "value": "On-chip | Offloaded | No"  
   }],  
 }
```

```
{
    "name": "TLS",
    "value": "On-chip | Offloaded | No"
}
],
"devices": [
    {
        "id": "<device-id1>",
        "connectivity": {
            "protocol": "uart",
            "serialPort": "<computer_serial_port_1>"
        },
        "identifiers": [
            {
                "name": "serialNo",
                "value": "<serialNo-value>"
            }
        ]
    },
    {
        "id": "<device-id2>",
        "connectivity": {
            "protocol": "uart",
            "serialPort": "<computer_serial_port_2>"
        },
        "identifiers": [
            {
                "name": "serialNo",
                "value": "<serialNo-value>"
            }
        ]
    }
]
}
```

ビルド、フラッシュ、テストを設定する

AWS IoT Device Tester にテストケースのビルドとボードへのフラッシュを実行させるためには、お使いのビルドおよびフラッシュツールのコマンドラインインターフェイスを使って AWS IoT Device Tester を設定しておく必要があります。ビルドとフラッシュの設定は、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。

1 つのデバイスをテストするための設定

`userdata.json` は次のような構造を持たなければなりません。

```
{
    "sourcePath": "<path-to-afr-source-code>",
    "buildTool": {
        "name": "<your-build-tool-name>",
        "version": "<your-build-tool-version>",
        "command": [
            "<your-build-script>"
        ]
    },
    "flashTool": {
        "name": "<your-flash-tool-name>",
        "version": "<your-flash-tool-version>",
        "command": [
            "<your-flash-script>"
        ]
    },
    "clientWifiConfig": {
        "wifiSSID": "<your-wifi-ssid>",
        " wifiPassword": "<your-wifi-password>",
        "wifiSecurityType": "<wifi-security-type>"
    },
    "testWifiConfig": {
        "wifiSSID": "<your-wifi-ssid>",
        " wifiPassword": "<your-wifi-password>",
        "wifiSecurityType": "<wifi-security-type>"
    }
}
```

```
        "wifiSecurityType": "<wifi-security-type>"  
    },  
    "otaConfiguration": {  
        "otaFirmwareFilePath": "<path-to-the-device-binary>",  
        "deviceFirmwareFileName": "<your-device-firmware-name>.bin",  
        "awsSignerPlatform": "AmazonFreeRTOS-Default",  
        "awsSignerCertificateArn": "arn:aws:acm:us-east-1:1000000001:certificate/e416379d-f3d6-46f3-868e-8721075ff076",  
        "awsUntrustedSignerCertificateArn": "arn:aws:acm:us-east-1:1000000001:certificate/0c81e2c6-f85e-46b1-9ed1-2c404309b210",  
        "awsSignerCertificateFileName": "ecdsa-sha256-signer.crt.pem",  
        "compileCodesignerCertificate": true  
    }  
}
```

次のリストは、`userdata.json` ファイルで使用される属性です。

sourcePath

移植された Amazon FreeRTOS ソースコードのルートへのパス。AWS IoT Device Tester は、`{{testData.sourcePath}}` 変数に値を格納します。

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。

Note

IDE を使用している場合は、ヘッドレスモードで実行するために IDE にコマンドラインを指定する必要があります。

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。

clientWifiConfig

クライアント側の Wi-Fi 設定。Wi-Fi ライブリテストは、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。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。クライアント側の Wi-Fi 設定は、`$AFR_HOME/tests/common/include/aws_clientcredential.h` で設定されます。次のマクロは、`aws_clientcredential.h` にある値を使って設定されます。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。

- `wifi_ssid`: C 文字列で表した Wi-Fi SSID。
- `wifi_password`: C 文字列で表した Wi-Fi パスワード。
- `wifiSecurityType`: 使用している Wi-Fi セキュリティの種類。

testWifiConfig

テストの Wi-Fi 設定。Wi-Fi ライブリテストは、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。この属性は、2 番目のアクセスポイントを設定します。テストの Wi-Fi 設定は、`$AFR_HOME/tests/common/wifi/aws_test_wifi.c` で設定されます。次のマクロは、`aws_test_wifi.c` にある値を使って設定されます。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。

- `testwifiWIFI_SSID`: C 文字列で表した Wi-Fi SSID。
- `testwifiWIFI_PASSWORD`: C 文字列で表した Wi-Fi パスワード。
- `testwifiWIFI_SECURITY`: 使用している Wi-Fi セキュリティの種類。次のいずれかの値になります。
 - `eWiFiSecurityOpen`
 - `eWiFiSecurityWEP`

- eWiFiSecurityWPA
 - eWiFiSecurityWPA2
- otaConfiguration**
- OTA 設定。
- otaFirmwareFilePath**
- ビルドの後に作成された OTA イメージの完全パス。
- deviceFirmwareFileName**
- MCU デバイスで、OTA ファームウェアがダウンロードされた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。
- awsSignerPlatform**
- OTA 更新ジョブの作成中に AWS Code Signer によって使用される署名アルゴリズム。現在、このフィールドで可能な値は、AmazonFreeRTOS-TI-CC3220SF と AmazonFreeRTOS-Default です。
- awsSignerCertificateArn**
- AWS Certificate Manager (ACM) 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) です。信頼された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。
- awsUntrustedSignerCertificateArn**
- ACM にアップロードされるコード署名証明書のうち、デバイスが信頼すべきでないものの Amazon リソースネーム (ARN)。これは、無効な証明書のテストケースをテストする際に使用されます。
- compileCodesignerCertificate**
- コード署名の検証証明書がプロジェクトにコンパイルする必要があるため、true に設定します。AWS IoT Device Tester は、ACM から信頼された証明書を取得し、それを `aws_codesigner_certificate.h` にコンパイルします。

複数の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、`userdata.json` ファイルで行われます。次の JSON の例は、複数のデバイスをテストするために AWS IoT Device Tester を設定する方法を示します。

```
{  
    "sourcePath": "<absolute-path-to/amazon-freertos>",  
    "buildTool": {  
        "name": "<your-build-tool-name>",  
        "version": "<your-build-tool-version>",  
        "command": [  
            "<absolute-path-to/build-parallel-script> {{testData.sourcePath}}"  
        ]  
    },  
    "flashTool": {  
        "name": "<your-flash-tool-name>",  
        "version": "<your-flash-tool-version>",  
        "command": [  
            "<absolute-path-to/flash-parallel-script> {{testData.sourcePath}}"  
            "{{device.connectivity.serialPort}}"  
        ]  
    }  
}
```

```

},
"clientWifiConfig": {
    "wifiSSID": "<ssid>",
    "wifiPassword": "<password>",
    "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA | eWiFiSecurityWPA2"
},
"testWifiConfig": {
    "wifiSSID": "<ssid>",
    "wifiPassword": "<password>",
    "wifiSecurityType": "eWiFiSecurityOpen | eWiFiSecurityWEP | eWiFiSecurityWPA | eWiFiSecurityWPA2"
},
"otaConfiguration": {
    "otaFirmwareFilePath": "{{testdata.sourcePath}}/<relative-path-to/ota-image-generated-in-build-process>",
    "deviceFirmwareFileName": "<absolute-path-to/ota-image-on-device>",
    "awsSignerPlatform": "AmazonFreeRTOS-Default | AmazonFreeRTOS-TI-CC3220SF",
    "awsSignerCertificateArn": "arn:aws:acm:<region>:<account-id>:certificate:<certificate-id>",
    "awsUntrustedSignerCertificateArn": "arn:aws:acm:<region>:<account-id>:certificate:<certificate-id>",
    "awsSignerCertificateFileName": "<awsSignerCertificate-file-name>",
    "compileCodesignerCertificate": true | false
}
}

```

次のリストは、`userdata.json` で使用される属性です。

`sourcePath`

移植された Amazon FreeRTOS ソースコードのルートへのパス。AWS IoT Device Tester は、`{{testdata.sourcePath}}` 変数に値を格納します。

`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。ビルドコマンドにあるソースコードパスへの参照は、すべて AWS IoT Device Tester 変数の `{{testdata.sourcePath}}` で置き換える必要があります。

`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にあるソースコードパスへの参照は、すべて AWS IoT Device Tester 変数の `{{testdata.sourcePath}}` で置き換える必要があります。

`clientWifiConfig`

クライアント側の Wi-Fi 設定。Wi-Fi ライブリテストは、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。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。クライアント側の Wi-Fi 設定は、\$AFR_HOME/tests/common/include/aws_clientcredential.h. で設定されます。次のマクロは、aws_clientcredential.h にある値を使って設定されます。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。

- `wifi_ssid`: C 文字列で表した Wi-Fi SSID。
- `wifi_password`: C 文字列で表した Wi-Fi パスワード。
- `wifiSecurityType`: 使用している Wi-Fi セキュリティの種類。

`testWifiConfig`

テストの Wi-Fi 設定。Wi-Fi ライブリテストは、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。この属性は、2 番目のアクセスポイントを設定します。テストの Wi-Fi 設

定は、\$AFR_HOME/tests/common/wifi/aws_test_wifi.c. で設定されます。次のマクロは、aws_test_wifi.c にある値を使って設定されます。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。

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 と testWifiConfig の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。

- testwifiWIFI_SSID: C 文字列で表した Wi-Fi SSID。
- testwifiWIFI_PASSWORD: C 文字列で表した Wi-Fi パスワード。
- testwifiWIFI_SECURITY: 使用している Wi-Fi セキュリティの種類。次のいずれかの値になります。
 - eWiFiSecurityOpen
 - eWiFiSecurityWEP
 - eWiFiSecurityWPA
 - eWiFiSecurityWPA2

otaConfiguration

OTA 設定。

otaFirmwareFilePath

ビルドの後に作成された OTA イメージの完全パス。たとえば、{{testData.sourcePath}}/ <relative-path/to/ota/image/from/source/root> と指定します。

deviceFirmwareFileName

OTA ファームウェアが配置されている MCU デバイス上の完全なファイルパス。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。

awsSignerPlatform

OTA 更新ジョブの作成中に AWS Code Signer によって使用される署名アルゴリズム。現在、このフィールドで可能な値は、AmazonFreeRTOS-TI-CC3220SF と AmazonFreeRTOS-Default です。

awsSignerCertificateArn

AWS Certificate Manager (ACM) 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) です。信頼された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。

awsUntrustedSignerCertificateArn

ACM にアップロードするコード署名証明書の ARN。

compileCodesignerCertificate

コード署名の検証証明書がプロジェクトにコンパイルする必要があるため、true に設定します。AWS IoT Device Tester は、ACM から信頼された証明書を取得し、それを aws_codesigner_certificate.h にコンパイルします。

AWS IoT Device Tester 変数

コードをビルドし、デバイスをフラッシュするコマンドは、デバイスに関する接続情報や他の情報がないと正しく実行できない場合があります。AWS IoT Device Tester では、JsonPath を使ってフラッシュ

コマンドおよびビルドコマンドでデバイス情報を参照することができます。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

AWS IoT Device Tester 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る変数。  
 {{ device.connectivity.serialPort }}  
  シリアルポートに拡張される変数。  
 {{ device.identifiers[?(@.name == 'serialNo')].value }}  
   デバイスのシリアル番号に拡張される変数。
```

AWS IoT Device Tester 変数と同時テスト

AWS IoT Device Tester 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 AWS IoT Device Tester の展開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中のソースコードパスは、testdata.sourcePath 変数を使って参照されなければなりません。AWS IoT Device Tester 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。

ファイルパスと Windows オペレーティングシステム

AWS IoT Device Tester を Windows で実行している場合は、AWS IoT Device Tester の config ファイル内のファイルパスにスラッシュ (/) を使用してください。たとえば、userdata.json の sourcePath は、c:/<dir1>/<dir2> と表記します。

Amazon FreeRTOS 適格性確認スイートの実行

AWS IoT Device Tester 実行可能ファイルを使って AWS IoT Device Tester を操作します。以下のコマンドラインは、デバイスピール (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。

```
devicetester_[linux | mac | win_x86-64] 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

AWS IoT Device テスターを Windows で実行している場合は、スラッシュ (/) を使用して userdata.json 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_x86-64] run-suite --suite-id AFQ_1 --group-id <group-id> --pool-id <pool-id> --userdata <userdata.json>
```

1 つのデバイスピールに対して 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ピールが 1 つしかない場合)、suite-id と pool-id は省略可能です。

AWS IoT Device Tester コマンドラインオプション

suite-id

オプション。実行するテストスイートを指定します。

pool-id

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

AWS IoT Device Tester コマンド

AWS IoT Device Tester コマンドは、以下のオペレーションをサポートしています。

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

デバイスピールに対してテストスイートを実行します。

結果とログ

このセクションでは、テストの結果とログを表示し、解釈する方法について説明します。

結果の表示

AWS IoT Device Tester は、適格性確認テストスイートを実行した後、1 回の実行ごとに `awsiotdevicetester_report.xml` と `AFO_Report.xml` の 2 つのレポートを生成します。これらのレポートは `<devicetester-extract-location>/results/<execution-id>` にあります。

`awsiotdevicetester_report.xml` は、AWS Partner Device Catalog にデバイスを出品する際に AWS に提出する適格性確認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。

- AWS IoT Device Tester のバージョン。
- テストされた Amazon FreeRTOS のバージョン。
- `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。

`AFO_Report.xml` は、標準の `junit.xml` 形式のレポートで、Jenkins や Bamboo など、既存の CI/CD プラットフォームに統合できます。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 (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。

AWS IoT Device Tester 結果の解釈

`awsiotdevicetester_report.xml` または `AFQ_Report.xml` 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>` は、テストの実行の概要を含みます。次に例を示します。

```
<testsuites name="AFQ 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`

AWS IoT Device Tester 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ない場合、そのデバイスは Amazon 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、適格性確認の証拠としてこのレポートを提出できます。

テストケースに障害やエラーがある場合は、`<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` という追加の属性があります。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>
```

<testcase> タグで使用される属性

name

テストケースの名前。

attempts

AWS IoT Device Tester がテストケースを実行した回数。

テストケースに障害やエラーが生じた場合、<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追加情報とともに) <testcase> タグに追加されます。次に例を示します。

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase" attempts="1">
  <failure type="Failure">Reason for the test case failure</failure>
  <error>Reason for the test case execution error</error>
</testcase>
```

ログの表示

テストの実行中に AWS IoT Device Tester 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`

AWS IoT Device Tester の Test Manager コンポーネントから生成されるログを含みます。たとえば、ログ関連の設定、テストの順序、レポートの生成などが含まれます。

`<test_group_name>.log` (for example, `Full_MQTT.log`)

テストグループのログ。テストされているデバイスからのログを含みます。

適合性再確認のためのテスト

AWS IoT Device Tester 適合性確認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、AWS IoT Device Tester を使ってマイクロコントローラーボードをテストすることができます。その後の適合性確認では、最新バージョンの Amazon FreeRTOS と AWS IoT Device Tester を用意してテストを再実行してください。

トラブルシューティング

Amazon FreeRTOS デバイスのトラブルシューティングテストを行うには、次のようなワークフローをお勧めします。

1. コンソール出力を読みます。
2. AFQ_Report.xml ファイルを確認します。
3. /results/<uuid>/logs にあるログファイルを確認します。
4. 以下の問題領域のいずれかを調べてください。
 - デバイス設定
 - デバイスインターフェイス
 - デバイツール
 - Amazon FreeRTOS ソースコード

デバイス設定のトラブルシューティング

AWS IoT Device Tester 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置かなければなりません。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。

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、`/results/<uuid>` ディレクトリの `AFO_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、各テストグループの `/results/<uuid>/<test-case-id>.log` を確認します。

解析エラー

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。AWS IoT Device Tester 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。

必須パラメータが見つからないエラー

AWS IoT Device Tester 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/<uuid>/logs` にある `<test_case_id>.log` ファイルに、欠けているパラメータが明示的に一覧されています。AWS IoT Device Tester は、JSON 設定ファイルのスキーマを検証し、サポートされている最新バージョンが使用されたことを確認します。

テストを開始できなかったエラー

テストの開始中の障害を示唆するエラーが生じことがあります。このエラーには、考えられる原因がいくつあるため、以下の領域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が正しい設定パラメータを持っていることを確認します。

デバイスインターフェイスとポート

このセクションでは、AWS IoT Device Tester 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

AWS IoT Device Tester 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.*
- Windows: COM*

デバイス ID 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。
- 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。マシンに接続されるデバイスの場合、デバイスをフラッシュするデータポートと、出力を読み取るためのポートの 2 つを備えているのが普通です。`device.json` ファイルで正しいポートを設定することが重要です。そうしないと、出力のフラッシュや読み取りに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データ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。

デバイステータの読み取り

AWS IoT Device Tester 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っ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ください。

- ポーレート - 115200
- テータビット - 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、ユーザーは何の設定もする必要がなく、AWS IoT Device Tester によって処理されます。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。

- Amazon 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。

Amazon FreeRTOS ソースコード

以下のセクションでは、Amazon FreeRTOS ソースコードの問題のトラブルシューティングを取り上げます。

コードの訂正一覧

Amazon FreeRTOS の各リリースには、訂正一覧を含むドキュメントが付属しています。このドキュメントは、`/amazon-freertos/tests` ディレクトリにあります。テストを実行する前にこのドキュメントに目を通すことをお勧めします。

訂正一覧のドキュメントには、以下のような理由でテストに失敗するデバイスが一覧されています。

- ハードウェアが特定の機能をサポートしていない。
- ハードウェアは機能をサポートしているが、Amazon FreeRTOS がデバイス上でその機能をまだサポートしていない。
- ハードウェアは機能をサポートしているが、基礎となるソフトウェアスタックがそのハードウェアをサポートしていない（非 AFR）。

訂正一覧に、デバイス固有の情報が含まれていない場合は、次のセクションでの説明に従ってデバッグポートを続けます。

Amazon FreeRTOS のデバッグ

ソースコードエラーが発生した場合、AWS IoT Device Tester は、`/results/<uuid>/logs` ディレクトリにある `<test-group-id>.log` ファイルにデバッグ出力を書き込みます。ファイルの中でエラーのインスタンスを検索します。エラーは、Amazon FreeRTOS ソースコードの中の場所を指しています。そこで、ログにある行番号とファイルパス情報を使用し、エラーの原因となったソースコードの部分を参照します。

ログ記録

AWS IoT Device Tester のログは 1箇所に配置されます。AWS IoT Device Tester のルートディレクトリから利用可能なファイルは以下のとおりです。

- `./results/<uuid>`
- `AFO_Report.xml`
- `awsiotdevicetester_report.xml`
- `/logs/<test_group_id>.log`

最も重要なログは、`<test_group_id>.log` と `results.xml` です。`results.xml` には、特定のエラーメッセージがどのテストで生成されたものかを示す情報が含まれます。それを見た後、もう一方のファイルで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 を実行すると、障害が短いメッセージと共にコンソールに報告されます。`<test_group_id>.log` でエラーの詳細を確認します。

ログエラー

`<test-group-id>.log` ファイルは、`/results/<uuid>` ディレクトリにあります。テスト実行には、それぞれ一意のテスト ID があり、この ID を使用して `<uuid>` ディレクトリが作成されます。テストグループ別のログは `<uuid>` ディレクトリにあります。AWS IoT コンソールを使用して失敗したテストグル

プを見つけています。次に `/results/<uuid>` ディレクトリで、そのグループのログファイルを開きます。
このファイルの情報には、フルビルドとフラッシュコマンドの出力、テスト実行の出力、詳細な AWS IoT
Device Tester コンソール出力が含まれます。

アクセス許可ポリシーテンプレート

AWS IoT Device Tester に必要なアクセス許可を付与するポリシーテンプレートは以下のとおりです。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "iam>CreatePolicy",
                "iam:DetachRolePolicy",
                "iam:DeleteRolePolicy",
                "s3>CreateBucket",
                "iam>DeletePolicy",
                "iam>CreateRole",
                "iam>DeleteRole",
                "iam:AttachRolePolicy",
                "s3>DeleteBucket",
                "s3:PutBucketVersioning"
            ],
            "Resource": [
                "arn:aws:s3:::idt*",
                "arn:aws:s3:::afr-ota*",
                "arn:aws:iam::*:policy/idt*",
                "arn:aws:iam::*:role/idt*"
            ]
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": [
                "iot>DeleteCertificate",
                "iot:AttachPolicy",
                "iot:DetachPolicy",
                "s3>DeleteObjectVersion",
                "iot>DeleteOTAUpdate",
                "s3:PutObject",
                "s3:GetObject",
                "iam:PassRole",
                "iot>DeleteStream",
                "iot>DeletePolicy",
                "iot:UpdateCertificate",
                "iot:GetOTAUpdate",
                "s3>DeleteObject",
                "iot:DescribeJobExecution",
                "s3:GetObjectVersion"
            ],
            "Resource": [
                "arn:aws:iot::*:thinggroup/idt*",
                "arn:aws:iot::*:policy/idt*",
                "arn:aws:iot::*:otaupdate/idt*",
                "arn:aws:iot::*:thing/idt*",
                "arn:aws:iot::*:cert/*",
                "arn:aws:iot::*:job/*",
                "arn:aws:iot::*:stream/*",
                "arn:aws:iam::*:role/idt*",
                "arn:aws:iot::*:topic/*"
            ]
        }
    ]
}
```

```
"arn:aws:s3:::afr-ota/*",
"arn:aws:s3:::idt/*",
"arn:aws:iam:::role/idt*"
],
},
{
"Sid": "VisualEditor2",
"Effect": "Allow",
"Action": [
"iot:DetachThingPrincipal",
"iot:AttachThingPrincipal",
"s3>ListBucketVersions",
"iot>CreatePolicy",
"iam>ListRoles",
"freertos>ListHardwarePlatforms",
"signer>DescribeSigningJob",
"s3>ListBucket",
"signer:*",
"iot>DescribeEndpoint",
"iot>CreateStream",
"signer>StartSigningJob",
"s3>ListAllMyBuckets",
"signer>ListSigningJobs",
"acm>GetCertificate",
"acm>ListCertificates",
"acm>ImportCertificate",
"freertos>DescribeHardwarePlatform",
"iot>CreateKeysAndCertificate",
"iot>CreateCertificateFromCsr",
"s3>GetBucketLocation",
"iot>GetRegistrationCode",
"iot>RegisterCACertificate",
"iot>RegisterCertificate",
"iot>UpdateCACertificate",
"iot>DeleteCACertificate",
"iot>DeleteCertificate",
"iot>UpdateCertificate"
],
"Resource": "*"
},
{
"Sid": "VisualEditor3",
"Effect": "Allow",
"Action": [
"s3>PutObject",
"s3>GetObject"
],
"Resource": [
"arn:aws:s3:::afr/*",
"arn:aws:s3:::idt/*"
]
},
{
"Sid": "VisualEditor4",
"Effect": "Allow",
"Action": [
"iot>CreateOTAUpdate",
"iot>CreateThing",
"iot>DeleteThing"
],
"Resource": "*"
}
]
```