aws

移植ガイド

FreeRTOS

FreeRTOS: 移植ガイド

Table of Contents

FreeRTOS の移植

FreeRTOS とは

世界をリードするチップ企業との 20 年間にわたる提携によって開発され、現在 170 秒ごとにダウンロードされている FreeRTOS は、マイクロコントローラーおよび小型マイクロプロセッサ向けの市場をリードするリアルタイムオペレーティングシステム (RTOS) です。MIT オープンソースライセンスで無料配布されている FreeRTOS には、すべての業種での使用に適したカーネルと増え続けるライブラリのセットが含まれています。FreeRTOS は、信頼性と使いやすさを重視して構築されています。FreeRTOS には、接続、セキュリティ、および無線通信 (OTA) に関する更新用のライブラリが含まれています。また、認定ボードで FreeRTOS の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。

詳細については、FreeRTOS.org にアクセスしてください。

FreeRTOS を IoT ボードに移植する

FreeRTOS ソフトウェアライブラリは、その機能とアプリケーションに基づいて、マイクロコントローラーベースのボードに移植する必要があります。

FreeRTOS をご利用のデバイスに移植する方法

1. 移植対象の FreeRTOS のダウンロード の手順に従って、移植対象である最新バージョンの FreeRTOS をダウンロードします。
2. 移植のためのワークスペースとプロジェクトの設定 の手順に従って、FreeRTOS ダウンロード内のファイルとフォルダを設定し、移植およびテストの準備をします。
3. FreeRTOS ライブラリの移植 の手順に従って、FreeRTOS ライブラリをご利用のデバイスに移植します。各移植トピックに、ポートをテストする手順が含まれています。

移植のよくある質問

FreeRTOS 移植とは

> FreeRTOS の移植とは、プラットフォームがサポートする必須の FreeRTOS ライブラリおよび FreeRTOS カーネル用の API について、ボード固有の実装を行うことです。移植により、API がボード上で動作するようになり、デバイスドライバーとプラットフォームベンダーにより提供さ

れる BSP との、必要な統合を実装します。移植には、ボードにより必要とされる設定の調整 (例: クロック率、スタックサイズ、ヒープサイズ) も含まれます。

このページや「FreeRTOS 移植ガイド」の残りの部分で回答が見つからない質問がある場合は、利用可能な FreeRTOS サポートオプションを参照してください。

移植対象の FreeRTOS のダウンロード

freertos.org から最新の FreeRTOS または長期サポート (LTS) バージョンをダウンロードするか、GitHub (FreeRTOS-LTS) または (FreeRTOS) からクローンを作成します。

> **Note**
>
> リポジトリを複製することをお勧めします。クローンすることで、メインブランチの更新がリポジトリにプッシュされたときに更新をピックアップすることが容易になります。

あるいは、FreeRTOS リポジトリまたは FreeRTOS-LTS リポジトリから個々のライブラリをサブモジュール化します。ただし、ライブラリのバージョンが FreeRTOS リポジトリまたは FreeRTOS-LTS リポジトリ内の `manifest.yml` ファイルにリストされている組み合わせと一致していることを確認してください。

FreeRTOS をダウンロードまたはクローン作成したら、ボードへの FreeRTOS ライブラリの移植を開始できます。手順については、移植のためのワークスペースとプロジェクトの設定を参照してから、FreeRTOS ライブラリの移植を参照してください。

移植のためのワークスペースとプロジェクトの設定

以下のステップに従って、ワークスペースとプロジェクトを設定します。

- 任意のプロジェクト構造とビルドシステムを使用して、FreeRTOS ライブラリをインポートします。
- ボードがサポートする統合開発環境 (IDE) とツールチェーンを使用して、プロジェクトを作成します。
- ボードサポートパッケージ (BSP) とボード固有のドライバーをプロジェクトに含めます。

ワークスペースを設定したら、個々の FreeRTOS ライブラリの移植を開始できます。

FreeRTOS ライブラリの移植

移植を開始する前に、「移植のためのワークスペースとプロジェクトの設定」の手順を実行します。

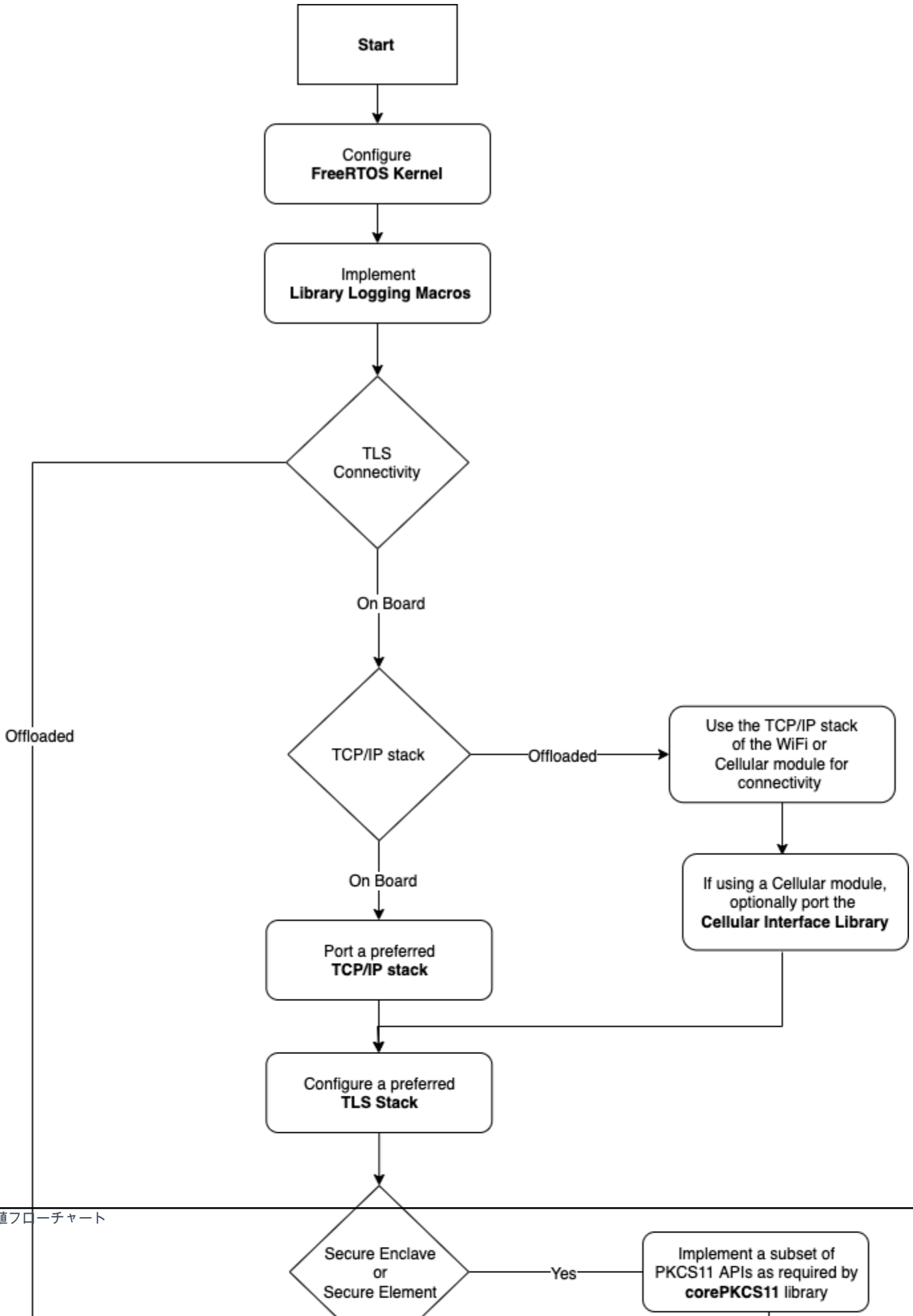
「FreeRTOS 移植フローチャート」では、移植に必要なライブラリについて説明しています。

FreeRTOS をデバイスに移植するには、次のトピックの指示に従います。

1. FreeRTOS カーネルの移植の設定
2. ライブラリロギングマクロの実装
3. TCP/IP スタックの移植
4. ネットワークトランスポートインターフェイスの移植
5. corePKCS11 ライブラリの移植
6. coreMQTT ライブラリの設定
7. coreHTTP ライブラリの設定
8. AWS IoT over-the-air (OTA) 更新ライブラリの移植
9. セルラーインターフェイスライブラリの移植

FreeRTOS 移植フローチャート

FreeRTOS をボードに移植する際には、以下のフローチャートを参考にしてください。


Start
Configure
FreeRTOS Kernel
Implement
Library Logging Macros
TLS
Connectivity
Offloaded
On Board
TCP/IP stack
Offloaded
Use the TCP/IP stack
of the WiFi or
Cellular module for
connectivity
If using a Cellular module,
optionally port the
Cellular Interface Library
On Board
Port a preferred
TCP/IP stack
Configure a preferred
TLS Stack
Secure Enclave
or
Secure Element
Yes
Implement a subset of
PKCS11 APIs as required by
corePKCS11 library

FreeRTOS カーネルの移植の設定

本セクションでは、FreeRTOS カーネルの移植を FreeRTOS 移植テストプロジェクトに統合する手順について説明します。利用可能なカーネル移植のリストについては、「FreeRTOS kernel ports」を参照してください。

FreeRTOS は、FreeRTOS カーネルを使用してマルチタスクおよびタスク間の通信を実現します。詳細については、「FreeRTOS ユーザーガイド」の「FreeRTOS カーネルの基礎」と FreeRTOS.org を参照してください。

> **Note**
>
> 新しいアーキテクチャへの FreeRTOS カーネルの移植については、このドキュメントで説明されていません。ご興味のある方は、FreeRTOS エンジニアリングチームにお問い合わせください。
> FreeRTOS 認定プログラムでは、既存の FreeRTOS カーネル移植のみがサポートされます。これらの移植に変更を加えることは、プログラム内では受け入れられません。詳細については、FreeRTOS カーネル移植ポリシーを確認してください。

前提条件

FreeRTOS カーネルを移植のためにセットアップするには、以下のものが必要です。

- 公式の FreeRTOS カーネル移植、またはターゲットプラットフォーム用に FreeRTOS がサポートする移植。
- ターゲットプラットフォーム用の正しい FreeRTOS カーネル移植ファイルおよびコンパイラが含まれている IDE プロジェクト。テストプロジェクトの設定については、「移植のためのワークスペースとプロジェクトの設定」を参照してください。

FreeRTOS カーネルの設定

FreeRTOS カーネルは、`FreeRTOSConfig.h` という設定ファイルを使用してカスタマイズされます。このファイルは、カーネル用のアプリケーション固有の設定を指定します。各設定オプションの説明については、FreeRTOS.org の「Customization」を参照してください。

デバイスと連携するように FreeRTOS カーネルを設定するには、`FreeRTOSConfig.h` をインクルードし、追加の FreeRTOS 設定を変更します。

各設定オプションの説明については、FreeRTOS.org の「Customization」設定を参照してください。

テスト

- 簡単な FreeRTOS タスクを実行して、メッセージをシリアル出力コンソールに記録します。
- メッセージが想定どおりにコンソールに出力されることを確認します。

ライブラリロギングマクロの実装

FreeRTOS ライブラリは、以下のロギングマクロを使用します。これらは詳細度の高い順にリストされています。

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

すべてのマクロの定義を提供する必要があります。推奨事項は以下のとおりです。

- マクロは C89 スタイルのロギングをサポートする必要があります。
- ロギングはスレッドセーフにする必要があります。複数のタスクからのログ行が相互にインターリーブしてはなりません。
- ロギング API によってブロックが発生してはなりません。アプリケーションタスクが I/O によってブロックされないようにする必要があります。

実装の詳細については、FreeRTOS.org の「Logging Functionality」を参照してください。この例で実装を確認できます。

テスト

- 複数のタスクを含むテストを実行して、ログがインターリーブしないことを確認します。
- テストを実行して、ロギング API が I/O によってブロックしていないことを確認します。
- C89,C99 スタイルのロギングなど、さまざまな標準でロギングマクロをテストします。

- Debug、Info、Error、Warning など、さまざまなログレベルを設定してロギングマクロをテストします。

TCP/IP スタックの移植

このセクションでは、オンボード TCP/IP スタックの移植とテストについて説明します。プラットフォームが TCP/IP および TLS 機能を別のネットワークプロセッサまたはモジュールにオフロードする場合は、この移植のセクションをスキップして ネットワークトランスポートインターフェイスの移植 を参照してください。

FreeRTOS+TCP は、FreeRTOS カーネル用のネイティブ TCP/IP スタックです。FreeRTOS+TCP は FreeRTOS エンジニアリングチームによって開発および管理されており、FreeRTOS で使用することをお勧めする TCP/IP スタックです。詳細については、「FreeRTOS+TCP の移植」を参照してください。または、サードパーティの TCP/IP スタック lwIP を使用することもできます。このセクションで説明するテスト手順では、TCP プレーンテキストのトランスポートインターフェイスのテストを使用するため、実装されている特定の TCP/IP スタックには依存しません。

FreeRTOS+TCP の移植

FreeRTOS+TCP は、FreeRTOS カーネル用のネイティブ TCP/IP スタックです。詳細については、「FreeRTOS.org」を参照してください。

前提条件

FreeRTOS+TCP ライブラリを移植するには、次のものが必要です。

- ベンダー提供のイーサネットまたは Wi-Fi ドライバーを含む IDE プロジェクト。

  テストプロジェクトの設定については、「移植のためのワークスペースとプロジェクトの設定」を参照してください。
- FreeRTOS カーネルの検証済み設定。

  ご使用のプラットフォーム用の FreeRTOS カーネルの設定については、「FreeRTOS カーネルの移植の設定」を参照してください。

移植

FreeRTOS+TCP ライブラリの移植を開始する前に、GitHub ディレクトリを調べて、ボードへの移植が既に存在するかどうか確認してください。

移植が存在しない場合は、次の操作を行います。

1. FreeRTOS.org の「FreeRTOS+TCP を別のマイクロコントローラーへ移植する」の手順に従って、FreeRTOS+TCP をデバイスに移植します。
2. 必要に応じて、FreeRTOS.org の「FreeRTOS+TCP を新しい Embedded C コンパイラへ移植する」の手順に従って、FreeRTOS+TCP を新しいコンパイラに移植します。
3. ベンダー提供のイーサネットまたは Wi-Fi ドライバーを使用する新しい移植を `NetworkInterface.c` というファイルに実装します。テンプレートについては、GitHub リポジトリを参照してください。

移植を作成済みであるか、移植が既に存在する場合は、`FreeRTOSIPConfig.h` を作成し、設定オプションをプラットフォームに適切な内容に設定します。設定オプションの詳細については、FreeRTOS.org の「FreeRTOS+TCP の設定」を参照してください。

テスト

FreeRTOS+TCP ライブラリを使用するか、サードパーティのライブラリを使用するかにかかわらず、以下の手順に従ってテストします。

- トランスポートインターフェイスのテストで、`connect/disconnect/send/receive` API の実装を提供します。
- プレーンテキスト TCP 接続モードでエコーサーバーをセットアップし、トランスポートインターフェイスのテストを実行します。

> **Note**
>
> FreeRTOS のデバイスを正式に認定するには、アーキテクチャで TCP/IP ソフトウェアスタックを移植する必要がある場合は、プレーンテキストの TCP 接続モードでトランスポートインターフェイステストに対してデバイスの移植されたソースコードを検証する必要があります AWS IoT Device Tester。FreeRTOS ユーザーガイド」の「Using AWS IoT Device Tester for FreeRTOS」の手順に従って、ポート検証 AWS IoT Device Tester 用に をセットアップします。 FreeRTOS 特定のライブラリのポートをテストするには、Device Tester `configs` フォルダの `device.json` ファイルで正しいテストグループを有効にする必要があります。

corePKCS11 ライブラリの移植

公開鍵暗号標準 #11 では、暗号化トークンを管理および使用するためのプラットフォームに依存しない API が定義されています。PKCS 11 は、標準と、標準で定義された API を指します。PKCS #11 暗号化 API は、キーストレージ、暗号化オブジェクトプロパティの取得/設定、およびセッションセマンティクスを抽象化します。これは、一般的な暗号オブジェクトの操作に広く使用されています。その機能により、アプリケーションソフトウェアは、暗号化オブジェクトをアプリケーションのメモリに公開することなく使用、作成、変更、削除することができます。

FreeRTOS ライブラリとリファレンス統合は、非対称キー、乱数生成、ハッシュなどの操作に重点を置いて、PCKS #11 インターフェイス標準のサブセットを使用します。以下の表は、ユースケースとサポートする必要のある PKCS #11 API の一覧です。

ユースケース

ユースケース	必要な PKCS #11 API ファミリー
すべて	初期化、完了、セッションのオープン/クローズ、GetSlotList、ログイン
プロビジョニング	GenerateKeyPair、CreateObject、DestroyObject、InitToken、GetTokenInfo
TLS	ランダム、署名、FindObject、GetAttributeValue
FreeRTOS+TCP	ランダム、ダイジェスト
OTA	検証、ダイジェスト、FindObject、GetAttributeValue

PKCS #11 モジュール全体を実装するタイミング

汎用のフラッシュメモリにプライベートキーを格納すると、評価やラピッドプロトタイピングのシナリオで便利になります。本稼働シナリオでは、データの盗難やデバイスの複製による脅威を軽減するために、専用の暗号化ハードウェアを使用することをお勧めします。暗号化ハードウェアには、暗号化シークレットキーのエクスポートを妨げる機能を備えたコンポーネントが含まれています。これをサポートするには、上の表で定義されているように、FreeRTOS ライブラリを操作するために必要な PKCS #11 のサブセットを実装する必要があります。

FreeRTOS corePKCS11 を使用するタイミング

corePKCS11 ライブラリには、Mbed TLS が提供する暗号化機能を使用する PKCS #11 インターフェイス (API) のソフトウェアベースの実装が含まれています。これは、ハードウェアに専用の暗号化ハードウェアがない場合のラピッドプロトタイピングや評価のシナリオのために用意されています。この場合、corePKCS11 PAL を実装するだけで、corePKCS11 ソフトウェアベースの実装がユーザーのハードウェアプラットフォームで動作するようになります。

corePKCS11 の移植

オンボードフラッシュメモリなどの不揮発性メモリ (NVM) への暗号化オブジェクトの読み書きには、実装が必要です。暗号化オブジェクトは、初期化されておらず、デバイスの再プログラミングで消去されない NVM のセクションに格納する必要があります。corePKCS11 ライブラリのユーザーは、認証情報を使用してデバイスをプロビジョニングしてから、こうした認証情報にアクセスする際に corePKCS11 インターフェイスを介する新しいアプリケーションを使用して、デバイスを再プログラムします。corePKCS11 PAL 移植は、以下を保存する場所を提供する必要があります。

- デバイスのクライアント証明書
- デバイスのクライアントのプライベートキー
- デバイスクのライアントのパブリックキー
- 信頼されたルート CA
- 安全なブートローダーと無線通信 (OTA) 更新用のコード検証パブリックキー (またはコード検証パブリックキーを含む証明書)
- ジャストインタイムのプロビジョニング証明書

ヘッダーファイルを含め、定義された PAL API を実装します。

PAL API

関数	説明
PKCS11_PAL_Initialize	PAL レイヤーを初期化します。初期化シーケンスの開始時に corePKCS11 ライブラリによって呼び出されます。
PKCS11_PAL_SaveObject	データを不揮発性ストレージに書き込みます。

関数	説明
PKCS11_PAL_FindObject	PKCS #11 CKA_LABEL を使用して、不揮発性ストレージ内の対応する PKCS #11 オブジェクトを検索し、存在する場合はそのオブジェクトのハンドルを返します。
PKCS11_PAL_GetObjectValue	ハンドルを指定して、オブジェクトの値を取得します。
PKCS11_PAL_GetObjectValueCleanup	PKCS11_PAL_GetObjectValue 呼び出しのクリーンアップ。PKCS11_PAL_GetObjectValue 呼び出しで割り当てられたメモリを解放するために使用できます。

テスト

FreeRTOS corePKCS11 ライブラリを使用するか、PKCS11 API の必要なサブセットを実装する場合は、FreeRTOS PKCS11 テストに合格する必要があります。これらは、FreeRTOS ライブラリに必要な関数が期待どおりに動作するかどうかをテストします。

このセクションでは、認定テストを使用して FreeRTOS PKCS11 テストをローカルで実行する方法についても説明します。

前提条件

FreeRTOS PKCS11 テストをセットアップするには、以下を実装する必要があります。

- サポートされている PKCS11 API のポート。
- FreeRTOS 認定の実装では、以下を含むプラットフォーム機能がテストされます。
  - FRTest_ThreadCreate
  - FRTest_ThreadTimedJoin
  - FRTest_MemoryAlloc
  - FRTest_MemoryFree

(GitHub の PKCS #11 の FreeRTOS ライブラリ統合テストについては、README.md ファイルを参照してください。)

移植テスト

- FreeRTOS-Libraries-Integration-Tests をサブモジュールとしてプロジェクトに追加します。サブモジュールは、ビルド可能な限り、プロジェクトのどのディレクトリにも配置できます。
- `config_template/test_execution_config_template.h` と `config_template/test_param_config_template.h` をビルドパス内のプロジェクトの場所にコピーし、名前を `test_execution_config.h` と `test_param_config.h` に変更します。
- 関連ファイルをビルドシステムに含めます。CMake を使用している場合は、`qualification_test.cmake` と `src/pkcs11_tests.cmake` を使用して関連ファイルを含めることができます。
- テスト出力ログとデバイスログがインターリーブしないように、`UNITY_OUTPUT_CHAR` を実装します。
- MbedTLS を統合し、これにより cryptoki オペレーションの結果を検証します。
- アプリケーションから `RunQualificationTest()` を呼び出します。

テストの設定

PKCS11 テストスイートは、PKCS11 実装に従って設定する必要があります。次の表は、PKCS11 テストに必要な `test_param_config.h` ヘッダーファイル内の設定を示しています。

PKSC11 テスト設定

設定	説明
PKCS11_TEST_RSA_KEY_SUPPORT	移植は RSA キーの機能をサポートします。
PKCS11_TEST_EC_KEY_SUPPORT	移植は EC キーの機能をサポートします。
PKCS11_TEST_IMPORT_PRIVATE_KEY_SUPPORT	移植はプライベートキーのインポートをサポートします。RSA キーと EC キーのインポートは、サポートしているキー機能が有効になっている場合にテストで検証されます。
PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT	移植はキーペア生成をサポートします。EC キーペア生成は、サポートしているキー機能が有効になっている場合にテストで検証されます。

設定	説明
PKCS11_TEST_PREPROVISIONED_SUPPORT	移植には認証情報が事前にプロビジョニングされています。認証情報の例としては、PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS 、PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS 、PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS が挙げられます。
PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS	テストで使用されるプライベートキーのラベル。
PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS	テストで使用されるパブリックキーのラベル。
PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS	テストで使用される証明書のラベル。
PKCS11_TEST_JITP_CODEVERIFY_ROOT_CERT_SUPPORTED	移植は JITP のストレージをサポートします。これを 1 に設定すると、JITP codeverify テストが有効になります。
PKCS11_TEST_LABEL_CODE_VERIFICATION_KEY	JITP codeverify テストで使用されるコード検証キーのラベル。
PKCS11_TEST_LABEL_JITP_CERTIFICATE	JITP codeverify テストで使用される JITP 証明書のラベル。
PKCS11_TEST_LABEL_ROOT_CERTIFICATE	JITP codeverify テストで使用されるルート証明書のラベル。

FreeRTOS ライブラリとリファレンス統合は、RSA キーや楕円曲線キーなどのキー機能設定のうち少なくとも 1 つと、PKCS11 API がサポートするキープロビジョニングメカニズムのうち少なくとも 1 つをサポートする必要があります。テストでは以下の設定を有効にする必要があります。

- 次のキー機能設定のうち少なくとも 1 つ:

  - PKCS11_TEST_RSA_KEY_SUPPORT
  - PKCS11_TEST_EC_KEY_SUPPORT
- 次のキープロビジョニング設定のうち少なくとも 1 つ:
  - PKCS11_TEST_IMPORT_PRIVATE_KEY_SUPPORT
  - PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT
  - PKCS11_TEST_PREPROVISIONED_SUPPORT

事前にプロビジョニングされたデバイス認証情報テストは、以下の条件で実行する必要があります。

- PKCS11_TEST_PREPROVISIONED_SUPPORT を有効にし、他のプロビジョニングメカニズムを無効にする必要があります。
- 有効になっているのは、PKCS11_TEST_RSA_KEY_SUPPORT または PKCS11_TEST_EC_KEY_SUPPORT のどちらか 1 つのキー機能だけです。
- ユーザーのキー機能に応じて、PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS、PKCS11_TEST_LABEL_DEVICE_PUBLI など、事前にプロビジョニングされたキーラベルを設定します。テストを実行する前に、これらの認証情報が存在している必要があります。

実装が事前にプロビジョニングされた認証情報やその他のプロビジョニングメカニズムをサポートしている場合、テストを異なる設定で複数回実行する必要がある場合があります。

> **Note**
>
> ラベル付きオブジェクト PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS、PKCS11_TEST_LABEL_DEVICE_PUBLI は、PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT または PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT のどちらか 1 つが有効になっている場合、テスト中に破棄されます。

テストを実行する

このセクションでは、認定テストを使用して PKCS11 インターフェイスをローカルでテストする方法について説明します。または、IDT を使用して実行を自動化することもできます。詳細について

は、「FreeRTOS ユーザーガイド」の「AWS IoT Device Tester for FreeRTOS」を参照してください。

次の手順では、テストの実行方法について説明します。

- `test_execution_config.h` を開いて、CORE_PKCS11_TEST_ENABLED を 1 に定義します。
- アプリケーションをビルドし、デバイスにフラッシュして実行します。テスト結果はシリアルポートに出力されます。

次は出力されたテスト結果の例です。

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

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

```
OK
```

すべてのテストに合格したら、テストは完了です。

> **Note**
>
> FreeRTOS のデバイスを正式に認定するには、デバイスの移植されたソースコードを で検証する必要があります AWS IoT Device Tester。FreeRTOS ユーザーガイド」の「Using AWS IoT Device Tester for FreeRTOS」の手順に従って、ポート検証 AWS IoT Device Tester 用にをセットアップします。特定のライブラリのポートをテストするには、 `configs`フォルダの `device.json` ファイル AWS IoT Device Tester で正しいテストグループを有効にする必要があります。

ネットワークトランスポートインターフェイスの移植

TLS ライブラリの統合

Transport Layer Security (TLS) 認証には、お好みの TLS スタックを使用してください。Mbed TLS は FreeRTOS ライブラリでテストされているため、使用することをお勧めします。この例は、この GitHub リポジトリにあります。

デバイスで使用されている TLS 実装にかかわらず、TLS スタックの基礎となるトランスポートフックを TCP/IP スタックで実装する必要があります。AWS IoTでサポートされている TLS 暗号スイートをサポートする必要があります。

ネットワークトランスポートインターフェイスライブラリの移植

coreMQTT と coreHTTP を使用するには、ネットワークトランスポートインターフェイスを実装する必要があります。ネットワークトランスポートインターフェイスには、1 つのネットワーク接続でデータを送受信するのに必要な関数ポインタとコンテキストデータが含まれています。詳細については、「Transport Interface」を参照してください。FreeRTOS には、これらの実装を検証するための一連の組み込みネットワークトランスポートインターフェイステストが用意されています。以下のセクションでは、これらのテストを実行するためのプロジェクトを設定する方法について説明します。

前提条件

このテストを移植するには、以下のことが必要です。

- 検証済みの FreeRTOS カーネルポートで FreeRTOS をビルドできるビルドシステムを備えたプロジェクト。
- ネットワークドライバーの実用的な実装。

移植

- FreeRTOS-Libraries-Integration-Tests をサブモジュールとしてプロジェクトに追加します。サブモジュールは、ビルド可能な限り、プロジェクトのどこに配置しても構いません。
- `config_template/test_execution_config_template.h` と `config_template/test_param_config_template.h` をビルドパス内のプロジェクトの場所にコピーし、名前を `test_execution_config.h` と `test_param_config.h` に変更します。
- 関連ファイルをビルドシステムに含めます。CMake を使用している場合は、`qualification_test.cmake` と `src/transport_interface_tests.cmake` を使用して関連ファイルを含めます。
- 以下の関数を適切なプロジェクトの場所に実装してください。
- • `network connect function`: 署名は、NetworkConnectFunc によって `src/common/network_connection.h` で定義されています。この関数は、ネットワークコンテキストへのポインタ、ホスト情報へのポインタ、およびネットワーク認証情報へのポインタを受け取ります。提供されたネットワーク認証情報を使用して、ホスト情報で指定されたサーバーとの接続を確立します。
  - `network disconnect function`: 署名は、NetworkDisconnectFunc によって `src/common/network_connection.h` で定義されています。この関数は、ネットワークコンテキストへのポインタを受け取ります。ネットワークコンテキストに保存されている以前に確立された接続を切断します。
  - `setupTransportInterfaceTestParam()`: これは `src/transport_interface/transport_interface_tests.h` で定義されています。実装には、`transport_interface_tests.h` で定義されているものとまったく同じ名前と署名が必要です。この関数は、TransportInterfaceTestParam 構造体へのポインタを受け取ります。この関数は、トランスポートインターフェイステストで使用される TransportInterfaceTestParam 構造体のフィールドに入力します。
- テスト出力ログがデバイスログとインターリーブしないように、UNITY_OUTPUT_CHAR を実装します。
- アプリケーションから `runQualificationTest()` を呼び出します。呼び出す前に、デバイスのハードウェアが正しく初期化され、ネットワークが接続されている必要があります。

認証情報管理 (デバイス上で生成されたキー)

`test_param_config.h` の FORCE_GENERATE_NEW_KEY_PAIR が 1 に設定されている場合、デバイスアプリケーションは新しいデバイス上のキーペアを生成し、パブリックキーを出力します。デバイスアプリケーションは、エコーサーバーとの TLS 接続を確立する際、エコーサーバーのルート CA およびクライアント証明書として、ECHO_SERVER_ROOT_CA および TRANSPORT_CLIENT_CERTIFICATE を使用します。IDT は、これらのパラメータを認定実行中に設定します。

認証情報管理 (キーのインポート)

デバイスアプリケーションは、エコーサーバーとの TLS 接続を確立する際、エコーサーバーのルート CA、クライアント証明書、およびクライアントプライベートキーとして、`test_param_config.h` の ECHO_SERVER_ROOT_CA、TRANSPORT_CLIENT_CERTIFICATE、および TRANSPORT_CLIENT_PRIVATE_KEY を使用します。IDT は、これらのパラメータを認定実行中に設定します。

テスト

このセクションでは、認定テストを使用してトランスポートインターフェイスをローカルでテストする方法について説明します。その他の詳細については、GitHub の FreeRTOS-Libraries-Integration-Tests の transport_interface セクションにある README.md ファイルを参照してください。

または、IDT を使用して実行を自動化することもできます。詳細については、「FreeRTOS ユーザーガイド」の「AWS IoT Device Tester for FreeRTOS」を参照してください。

テストの有効化

`test_execution_config.h` を開いて、TRANSPORT_INTERFACE_TEST_ENABLED を 1 に定義します。

テスト用エコーサーバーのセットアップ

ローカルテストには、テストを実行するデバイスからアクセスできるエコーサーバーが必要です。トランスポートインターフェイスの実装が TLS をサポートしている場合、エコーサーバーは TLS をサポートしている必要があります。まだ所有していない場合は、FreeRTOS-Libraries-Integration-Tests GitHub リポジトリにエコーサーバーの実装があります。

テスト用プロジェクトの設定

`test_param_config.h` で、ECHO_SERVER_ENDPOINT と ECHO_SERVER_PORT を前のステップのエンドポイントとサーバーのセットアップに更新します。

認証情報のセットアップ (デバイス上で生成されたキー)

- ECHO_SERVER_ROOT_CA をエコーサーバーのサーバー証明書に設定します。
- FORCE_GENERATE_NEW_KEY_PAIR を 1 に設定すると、キーペアが生成され、パブリックキーが取得されます。
- キー生成後に、FORCE_GENERATE_NEW_KEY_PAIR を 0 に戻します。
- パブリックキー、サーバーキー、および証明書を使用して、クライアント証明書を生成します。
- TRANSPORT_CLIENT_CERTIFICATE を生成されたクライアント証明書に設定します。

認証情報のセットアップ (キーのインポート)

- ECHO_SERVER_ROOT_CA をエコーサーバーのサーバー証明書に設定します。
- TRANSPORT_CLIENT_CERTIFICATE を事前に生成されたクライアント証明書に設定します。
- TRANSPORT_CLIENT_PRIVATE_KEY を事前に生成されたクライアントプライベートキーに設定します。

アプリケーションのビルドとフラッシュ

任意のツールチェーンを使用して、アプリケーションのビルドとフラッシュを行います。`runQualificationTest()` が呼び出されると、トランスポートインターフェイスのテストが実行されます。テスト結果はシリアルポートに出力されます。

> **Note**
>
> FreeRTOS のデバイスを正式に認定するには、デバイスの移植されたソースコードを OTA PAL および OTA E2E テストグループに対して検証する必要があります AWS IoT Device Tester。FreeRTOS ユーザーガイド」の「Using AWS IoT Device Tester for FreeRTOS」の手順に従って、ポート検証 AWS IoT Device Tester 用に をセットアップします。 FreeRTOS 特定のライブラリのポートをテストするには、 フォルダの `device.json` ファイル AWS IoT Device Tester `configs`で正しいテストグループを有効にする必要があります。

coreMQTT ライブラリの設定

エッジ上のデバイスは、MQTT プロトコルを使用して AWS クラウドと通信できます。 AWS IoT は、エッジの接続されたデバイスとの間でメッセージを送受信する MQTT ブローカーをホストします。

coreMQTT ライブラリは、FreeRTOS が動作するデバイス用の MQTT プロトコルを実装します。coreMQTT ライブラリを移植する必要はありませんが、資格認定を得るには、デバイスのテストプロジェクトをすべての MQTT テストに合格させる必要があります。詳細については、FreeRTOS ユーザーガイドの coreMQTT ライブラリを参照してください。

前提条件

coreMQTT ライブラリのテストをセットアップするには、ネットワークトランスポートインターフェイスの移植が必要です。詳細については、「ネットワークトランスポートインターフェイスの移植」を参照してください。

テスト

coreMQTT 統合テストを実行します。

- クライアント証明書を MQTT ブローカーに登録します。
- ブローカーのエンドポイントを `config` で設定し、統合テストを実行します。

リファレンス MQTT デモの作成

すべての MQTT 操作のスレッドセーフを処理するために、coreMQTT エージェントを使用することが推奨されます。また、ユーザーがアプリケーションに TLS、MQTT、その他の FreeRTOS ライブラリが効果的に統合されているかどうかを検証するには、パブリッシュタスクとサブスクライブタスク、および Device Advisor テストも必要です。

FreeRTOS のデバイスを正式に認定するには、 AWS IoT Device Tester MQTT テストケースを使用して統合プロジェクトを検証します。セットアップとテストの手順については、「AWS IoT Device Advisor workflow」を参照してください。TLS と MQTT の必須テストケースは以下のとおりです。

TLS テストケース

テストケース	テストケース	必須のテスト
TLS	TLS Connect	はい
TLS	TLS サポート AWS IoT 暗号スイート	推奨される暗号スイート
TLS	TLS 非セキュアサーバー証明書	はい
TLS	TLS の不正なサブジェクト名のサーバー証明書	はい

MQTT テストケース

テストケース	テストケース	必須のテスト
MQTT	MQTT 接続	はい
MQTT	MQTT 接続ジッター再試行	Yes、警告なし
MQTT	MQTT サブスクライブ	はい
MQTT	MQTT パブリッシュ	はい
MQTT	MQTT ClientPuback QoS1	はい
MQTT	MQTT No Ack PingResp	はい

coreHTTP ライブラリの設定

エッジ上のデバイスは、HTTP プロトコルを使用して AWS 、エッジの接続されたデバイスとの間でメッセージを送受信する HTTP サーバーを Cloud. AWS IoT services ホストと通信できます。

テスト

テストを行うには、以下のステップを実行します。

- AWS または HTTP サーバーによる TLS 相互認証用に PKI を設定します。
- coreHTTP 統合テストを実行します。

AWS IoT over-the-air (OTA) 更新ライブラリの移植

FreeRTOS 無線通信経由 (OTA) 更新を使用すると、次の操作を実行できます。

- 新しいファームウェアイメージを単一のデバイス、デバイスのグループ、またはフリート全体に展開します。
- グループに追加、リセット、または再プロビジョニングされると、デバイスにファームウェアを展開します。
- 新しいファームウェアがデバイスに導入された後、そのファームウェアの信頼性と完全性を検証します。
- デプロイの進行状況をモニタリングします。
- 失敗したデプロイをデバッグします。
- Code Signing for を使用してファームウェアにデジタル署名します AWS IoT。

詳細については、「FreeRTOS ユーザーガイド」の「FreeRTOS 無線通信経由更新」と AWS IoT 無線通信経由更新に関するドキュメントを参照してください。

OTA 更新ライブラリを使用して、OTA の機能を FreeRTOS アプリケーションに統合できます。詳細については、FreeRTOS ユーザーガイドの FreeRTOS OTA 更新ライブラリを参照してください。

FreeRTOS デバイスは、受信した OTA ファームウェアイメージに対して暗号化コード署名検証を実行する必要があります。以下のアルゴリズムを推奨します。

- 楕円曲線デジタル署名アルゴリズム (ECDSA)
- NIST P256 曲線
- SHA-256 ハッシュ

前提条件

- 移植のためのワークスペースとプロジェクトの設定 の手順を完了させます。
- ネットワークトランスポートインターフェイスポートを作成します。

詳細については、ネットワークトランスポートインターフェイスの移植 を参照してください。

- coreMQTT ライブラリを統合します。詳細については、「FreeRTOS ユーザーガイド」の「coreMQTT ライブラリ」を参照してください。
- OTA 更新をサポートするブートローダーを作成します。

プラットフォームの移植

OTA ライブラリを新しいデバイスに移植するには、OTA ポータブル抽象化レイヤー (PAL) を実装する必要があります。PAL API は、実装固有の詳細を指定する必要がある ota_platform_interface.h ファイルで定義されています。

関数名	説明
`otaPal_Abort`	OTA 更新を停止させます。
`otaPal_CreateFileForRx`	受信したデータチャンクを保存するファイルを作成します。
`otaPal_CloseFile`	指定されたファイルを閉じます。暗号保護を実装するストレージを使用している場合、これによってファイルが認証される可能性があります。
`otaPal_WriteBlock`	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。成功すると、関数によって書き込まれたバイト数が返されます。その他の場合は、関数によって負のエラーコードが返されます。ブロックのサイズは常に 2 の累乗でアラインされます。詳細については、OTA ライブラリの「Configurations」を参照してください。
`otaPal_ActivateNewImage`	新しいファームウェアイメージをアクティブ化または起動します。一部のポートでは、デバイスがプログラム的に同期的にリセットされると、この関数は戻りません。

関数名	説明
otaPal_SetPlatformImageState	最新の OTA 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ものを行います。この関数を実装するには、ボード (プラットフォーム) の詳細とアーキテクチャについてのドキュメントを参照してください。
otaPal_GetPlatformImageState	OTA 更新イメージの状態を取得します。

デバイスがそれらをサポートしている場合は、この表の関数を実装してください。

関数名	説明
otaPal_CheckFileSignature	指定したファイルの署名を確認します。
otaPal_ReadAndAssumeCertificate	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。
otaPal_ResetDevice	デバイスをリセットします。

> **Note**
>
> OTA 更新をサポートできるブートローダーがあることを確認してください。 AWS IoT デバイスブートローダーの作成方法については、「IoT デバイスブートローダー」を参照してください。

E2E テストと PAL テスト

OTA PAL テストと E2E テストを実行します。

E2E テスト

OTA エンドツーエンド (E2E) テストは、デバイスの OTA 機能を検証し、実際のシナリオをシミュレートするために使用されます。このテストにはエラー処理が含まれます。

前提条件

このテストを移植するには、以下のことが必要です。

- OTA AWS ライブラリが統合されたプロジェクト。追加情報については、OTA ライブラリの「移植ガイド」を参照してください。
- OTA ライブラリを使用してデモアプリケーションを移植し、 AWS IoT Core とやり取りして OTA 更新を行います。「OTA デモアプリケーションの移植」を参照してください。
- IDT ツールをセットアップします。これにより、OTA E2E ホストアプリケーションが実行され、さまざまな構成でデバイスをビルド、フラッシュ、監視し、OTA ライブラリの統合が検証されます。

OTA デモアプリケーションの移植

OTA E2E テストには、OTA ライブラリの統合を検証するための OTA デモアプリケーションが必要です。デモアプリケーションには、OTA ファームウェアの更新を実行できる能力が必要です。FreeRTOS OTA デモアプリケーションは FreeRTOS GitHub リポジトリにあります。デモアプリケーションをリファレンスとして使用し、仕様に従って変更することをお勧めします。

移植手順

1. OTA エージェントを初期化します。
2. OTA アプリケーションコールバック関数を実装します。
3. OTA エージェントイベント処理タスクを作成します。
4. OTA エージェントを開始します。
5. OTA エージェントの統計を監視します。
6. OTA エージェントをシャットダウンします。

詳細な手順については、「FreeRTOS OTA over MQTT - Entry point of the demo」を参照してください。

設定

を操作するには、次の設定が必要です AWS IoT Core。

- AWS IoT Core クライアント認証情報

  - Amazon 信頼サービスエンドポイントを使用して、`Ota_Over_Mqtt_Demo/demo_config.h` の democonfigROOT_CA_PEM をセットアップします。詳細については、AWS サーバー認証を参照してください。
  - AWS IoT クライアント認証情報`Ota_Over_Mqtt_Demo/demo_config.h`を使用して、 で democonfigCLIENT_CERTIFICATE_PEM と democonfigCLIENT_PRIVATE_KEY_PEM を設定します。クライアント証明書とプライベートキーについては、AWS クライアント認証の詳細を参照してください。
- アプリケーションバージョン
- OTA コントロールプロトコル
- OTA データプロトコル
- コード署名の認証情報
- その他の OTA ライブラリ設定

前述の情報は、FreeRTOS OTA デモアプリケーションの `demo_config.h` および `ota_config.h` にあります。詳細については、「FreeRTOS OTA over MQTT - Setting up the device」を参照してください。

ビルド検証

デモアプリケーションを実行して OTA ジョブを実行します。テストが正常に完了したら、引き続き OTA E2E テストを実行できます。

FreeRTOS OTA デモでは、FreeRTOS Windows Simulator での OTA クライアントと AWS IoT Core OTA ジョブの設定に関する詳細情報を提供します。 AWS OTA は MQTT プロトコルと HTTP プロトコルの両方をサポートしています。詳細については、次の例を参照してください。

- Windows Simulator での MQTT 経由の OTA デモ
- Windows Simulator での HTTP 経由の OTA デモ

IDT ツールによるテストの実行

OTA E2E テストを実行するには、 AWS IoT Device Tester (IDT) を使用して実行を自動化する必要があります。詳細については、「FreeRTOS ユーザーガイド」の「AWS IoT Device Tester for FreeRTOS」を参照してください。

E2E テストケース

テストケース	説明
OTAE2EGreaterVersion	定期的な OTA 更新のハッピーパステスト。これにより、新しいバージョンで更新が作成され、デバイスが正常に更新されます。
OTAE2EBackToBackDownloads	このテストでは、3 回連続で OTA 更新を作成します。デバイスは 3 回連続して更新される見込みです。
OTAE2ERollbackIfUnableToConnectAfterUpdate	このテストでは、デバイスが新しいファームウェアでネットワークに接続できない場合に、以前のファームウェアにロールバックすることを確認します。
OTAE2ESameVersion	このテストでは、バージョンが変わらない場合に、受信したファームウェアをデバイスが拒否することを確認します。
OTAE2EUnsignedImage	このテストでは、イメージが署名されていない場合に、デバイスが更新を拒否することを確認します。
OTAE2EUntrustedCertificate	このテストでは、ファームウェアが信頼できない証明書で署名されている場合に、デバイスが更新を拒否することを確認します。
OTAE2EPreviousVersion	このテストでは、デバイスが古い更新のバージョンを拒否することを確認します。
OTAE2EIncorrectSigningAlgorithm	デバイスによって、サポートされる署名アルゴリズムとハッシュアルゴリズムは異なります。このテストでは、サポートされていないアルゴリズムで作成されている場合に、デバイスが OTA 更新に失敗することを確認します。

テストケース	説明
OTAE2EDisconnectResume	これは一時停止/再開機能のハッピーパステストです。このテストでは、OTA 更新を作成して、更新を開始します。次に、同じクライアント ID (モノの名前) と認証情報 AWS IoT Core を使用して に接続します。 AWS IoT Core その後、 はデバイスを切断します。デバイスは、切断されていることを検出し AWS IoT Core、一定期間後に自身を停止状態に移行して、ダウンロードへの再接続 AWS IoT Core と再開を試みることが期待されます。
OTAE2EDisconnectCancelUpdate	このテストでは、一時停止状態のときに OTA ジョブがキャンセルされた場合に、デバイスが自動的に復旧できるかどうかを確認します。OTAE2EDisconnectResume テストと同じことを行いますが、接続後にデバイスを切断 AWS IoT Coreすると、OTA 更新がキャンセルされます。新しい更新が作成されます。デバイスは に再接続し AWS IoT Core、現在の更新を中止して待機状態に戻して、次の更新を受け入れて完了することが期待されます。
OTAE2EPresignedUrlExpired	OTA 更新が作成されたら、S3 の署名付き URL の有効期間を設定できます。このテストでは、URL の有効期限が切れてダウンロードを完了できなくても、デバイスが OTA を実行できることを確認します。デバイスは、新しいジョブドキュメントを要求することが期待されます。このドキュメントには、ダウンロードを再開するための新しい URL が含まれています。

テストケース	説明
OTAE2E2UpdatesCancel1st	このテストでは、2 回続けて OTA 更新を作成します。デバイスが最初の更新をダウンロードしていると報告すると、テストは最初の更新を強制的にキャンセルします。デバイスは、現在の更新を中止して 2 回目の更新を取得し、更新を完了することが期待されます。
OTAE2ECancelThenUpdate	このテストでは、2 回続けて OTA 更新を作成します。デバイスが最初の更新をダウンロードしていると報告すると、テストは最初の更新を強制的にキャンセルします。デバイスは、現在の更新を中止して 2 回目の更新を取得し、更新を完了することが期待されます。
OTAE2EImageCrashed	このテストでは、イメージがクラッシュしたときに、デバイスが更新を拒否できることを確認します。

PAL テスト

前提条件

ネットワークトランスポートインターフェイスのテストを移植するには、次のものが必要です。

- 有効な FreeRTOS カーネルポートを使用して FreeRTOS をビルドできるプロジェクト。
- OTA PAL の実用的な実装。

移植

- FreeRTOS-Libraries-Integration-Tests をサブモジュールとしてプロジェクトに追加します。プロジェクト内のサブモジュールの場所は、そのサブモジュールをビルドできる場所でなければなりません。
- `config_template/test_execution_config_template.h` と `config_template/test_param_config_template.h` をビルドパス内の場所にコピーし、名前を `test_execution_config.h` と `test_param_config.h` に変更します。

- 関連ファイルをビルドシステムに含めます。CMake を使用している場合は、qualification_test.cmake と src/ota_pal_tests.cmake を使用して関連ファイルを含めることができます。
- 次の関数を実装して、テストを設定します。
  - SetupOtaPalTestParam(): src/ota/ota_pal_test.h で定義されています。実装には、ota_pal_test.h で定義されているものと同じ名前と署名が必要です。現在のところ、この機能を設定する必要はありません。
- テスト出力ログがデバイスログとインターリーブしないように、UNITY_OUTPUT_CHAR を実装します。
- アプリケーションから RunQualificationTest() を呼び出します。呼び出す前に、デバイスのハードウェアが正しく初期化され、ネットワークが接続されている必要があります。

テスト

このセクションでは、OTA PAL 認定テストのローカルテストについて説明します。

テストの有効化

test_execution_config.h を開いて、OTA_PAL_TEST_ENABLED を 1 に定義します。

test_param_config.h で、以下のオプションを更新します。

- OTA_PAL_TEST_CERT_TYPE: 使用する証明書タイプを選択します。
- OTA_PAL_CERTIFICATE_FILE: デバイス証明書へのパス (該当する場合)。
- OTA_PAL_FIRMWARE_FILE: ファームウェアファイルの名前 (該当する場合)。
- OTA_PAL_USE_FILE_SYSTEM: OTA PAL がファイルシステム抽象化を使用している場合は、1 に設定します。

任意のツールチェーンを使用して、アプリケーションのビルドとフラッシュを行います。RunQualificationTest() が呼び出されると、OTA PAL テストが実行されます。テスト結果はシリアルポートに出力されます。

OTA タスクの統合

- 現在の MQTT デモに OTA エージェントを追加します。
- で OTA エンドツーエンド (E2E) テストを実行します AWS IoT。これにより、統合が期待どおりに機能しているかどうかが検証されます。

> **Note**
>
> FreeRTOS のデバイスを正式に認定するには、デバイスの移植されたソースコードを OTA PAL および OTA E2E テストグループに対して検証する必要があります AWS IoT Device Tester。FreeRTOS ユーザーガイド」の「Using AWS IoT Device Tester for FreeRTOS」の指示に従って、ポート検証 AWS IoT Device Tester 用に を設定します。 FreeRTOS 特定のライブラリのポートをテストするには、 フォルダの `device.json` ファイル AWS IoT Device Tester `configs`で正しいテストグループを有効にする必要があります。

IoT デバイスブートローダー

独自のセキュアなブートローダーアプリケーションを提供する必要があります。設計と実装がセキュリティの脅威を適切に緩和するようにしてください。参考までに、脅威モデリングを以下に示します。

IoT デバイスブートローダーの脅威モデリング

背景

実用的な定義として、この脅威モデルによって参照される組み込み AWS IoT デバイスは、クラウドサービスとやり取りするマイクロコントローラーベースの製品です。コンシューマー、商用、または産業用にデプロイできます。IoT デバイスは、ユーザー、患者、マシン、環境に関するデータを収集し、電球やドアロックから工場の機械に至るまで、あらゆるものを制御できます。

脅威モデリングは、仮定の敵対者の観点からのセキュリティへのアプローチです。敵対者の目標と方法を検討することで、脅威のリストが作成されます。脅威とは、敵によって実行されるリソースまたはアセットに対する攻撃です。リストは優先順位付けされ、緩和策を識別および作成するために使用されます。緩和策を選択する場合、実装と保守のコストは、それらが提供する実際のセキュリティ価値とのバランスを取る必要があります。複数の 脅威モデルの方法論があります。各 は、安全で成功した AWS IoT 製品の開発をサポートできます。

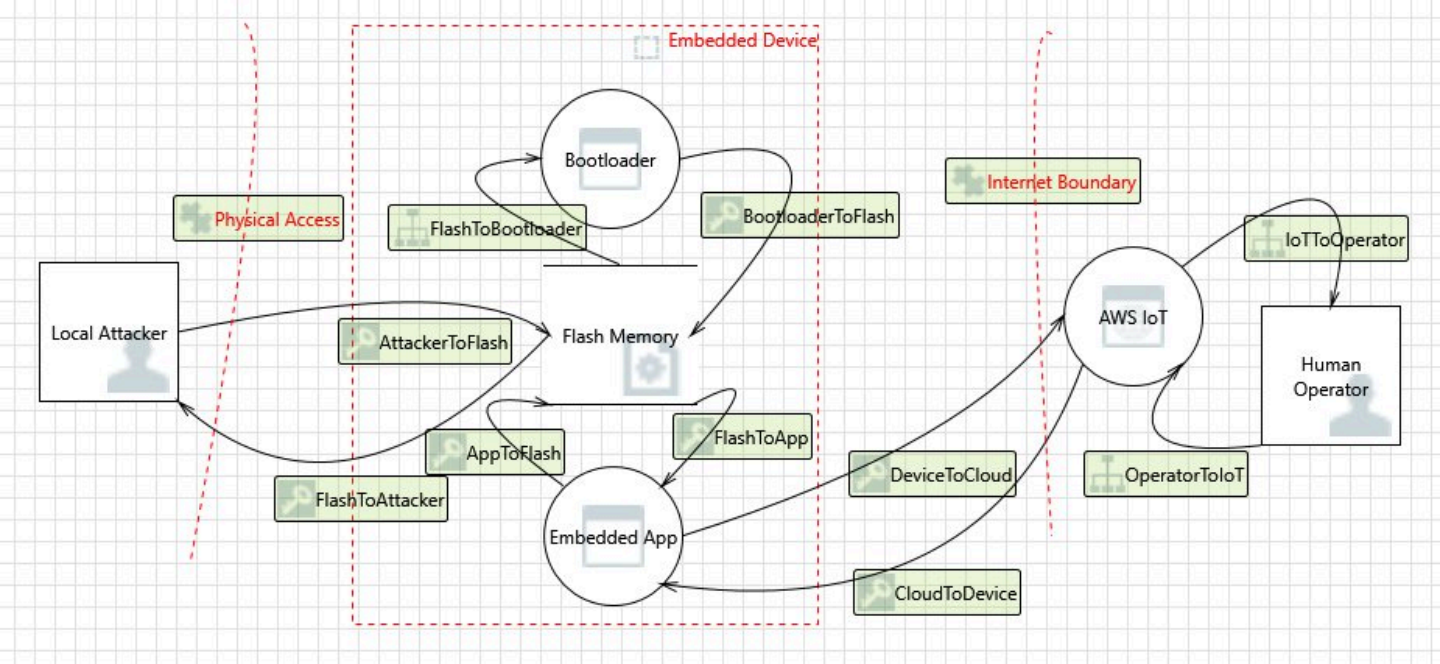
FreeRTOS は、 AWS IoT デバイスに OTA (over-the-airソフトウェア更新を提供します。更新機能は、クラウドサービスとオンデバイスソフトウェアライブラリとパートナー提供のブートローダーを組み合わせます。この脅威モデルは、特にブートローダーに対する脅威に焦点を当てています。

ブートローダーのユースケース

- 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デプロイします。
- 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。
- デバイスは、信頼できるソースから変更されていないソフトウェアのみを実行します。
- デバイスは、OTA を介して受け取ったソフトウェアに障害耐性があります。

データフロー図



脅威

攻撃によっては、複数の緩和モデルがあります。例えば、悪意のあるファームウェアイメージを配信することを意図したネットワーク中間者は、TLS サーバーによって提供される証明書と新しいファームウェアイメージのコード署名者証明書の両方の信頼を検証することで軽減されます。ブートローダーのセキュリティを最大化するために、ブートローダー以外の緩和策は信頼できないと見なされます。ブートローダーには、攻撃ごとに組み込み緩和策が必要です。多層化された緩和策は、多重防御と呼ばれます。

脅威:

- 攻撃者は、悪意のあるファームウェアイメージを配信するために、デバイスからサーバーへの接続をハイジャックします。

緩和の例

- ブートローダーは、起動時に、既知の証明書を使用してイメージの暗号化署名を検証します。検証に失敗した場合、ブートローダーは前のイメージにロールバックします。

- 攻撃者は、バッファオーバーフローを悪用して、フラッシュに保存されている既存のファームウェアイメージに悪意のある動作を導入します。

  緩和の例

  - ブート時に、前に説明したように、ブートローダーが検証します。以前のイメージが使用できない状態で検証に失敗すると、ブートローダーは停止します。
  - ブート時に、前に説明したように、ブートローダーが検証します。以前のイメージが使用できない状態で検証に失敗すると、ブートローダーはフェイルセーフ OTA 専用モードに入ります。

- 攻撃者は、以前に保存されたイメージからデバイスを起動します。このイメージは悪用可能です。

  緩和の例

  - 最後のイメージを保存する Flash セクターは、新しいイメージのインストールとテストに成功すると消去されます。
  - ヒューズはアップグレードが成功するたびに焼き付けられ、正しい数のヒューズが焼き付けられていない限り、各イメージの実行は拒否されます。

- OTA 更新は、デバイスをブロックする障害のあるイメージや悪意のあるイメージを配信します。

  緩和の例

  - ブートローダーは、前のイメージへのロールバックをトリガーするハードウェアウォッチドッグタイマーを開始します。

- 攻撃者はブートローダーにパッチを適用してイメージの検証をバイパスし、デバイスが署名されていないイメージを受け入れるようにします。

  緩和の例

  - ブートローダーは ROM (読み取り専用メモリ) にあり、変更することはできません。
  - ブートローダーは OTP (ワンタイムプログラム可能なメモリ) にあり、変更することはできません。
  - ブートローダーは ARM TrustZone のセキュアゾーンにあり、変更することはできません。

- デバイスが悪意のあるイメージを受け入れるように、攻撃者は検証証明書を置き換えます。

緩和の例

- 証明書は暗号コプロセッサにあり、変更することはできません。
- 証明書は ROM (または OTP、またはセキュアゾーン) にあり、変更することはできません。

脅威のさらなるモデル化

この脅威モデルでは、ブートローダーのみを考慮します。さらに脅威モデリングを行うことで、全体的なセキュリティを向上させることができます。推奨される方法は、敵対者の目標、それらの目標の対象となるアセット、およびアセットへの参入ポイントをリストすることです。脅威のリストは、エントリポイントに対する攻撃を考慮してアセットをコントロールすることで作成できます。以下は、IoT デバイスの目標、アセット、エントリポイントの例です。これらのリストはすべてを網羅したものではなく、さらなる検討を促進することを目的としています。

敵対者の目標

- 資金の強要
- 評判を落とす
- データの改ざん
- リソースの転送
- ターゲットをリモートでスパイ
- サイトへの物理的なアクセスを獲得する
- 大惨事
- 恐怖を与える

主要なアセット

- プライベートキー
- クライアント証明書
- CA ルート証明書
- セキュリティ認証情報とトークン
- お客様の個人識別情報
- 企業シークレットの実装
- センサーデータ

- クラウド分析データストア
- クラウドインフラストラクチャ

エントリポイント

- DHCP レスポンス
- DNS レスポンス
- MQTT over TLS
- HTTPS レスポンス
- OTA ソフトウェアイメージ
- その他 (アプリケーションによって指示される USB など)
- バスへの物理的なアクセス
- デキャップされた IC

セルラーインターフェイスライブラリの移植

FreeRTOS は、TCP オフロードが実行されたセルラー抽象化レイヤーの AT コマンドをサポートしています。詳細については、freertos.org のセルラーインターフェイスライブラリとセルラーインターフェイスライブラリの移植を参照してください。

前提条件

セルラーインターフェイスライブラリに直接的な依存関係はありません。ただし、FreeRTOS ネットワークスタック内でイーサネット、Wi-Fi、セルラーを共存させることはできないため、開発者は、そのうち 1 つを選択してネットワークトランスポートインターフェイスの移植と統合させる必要があります。

> **Note**
>
> セルラーモジュールが TLS オフロードをサポートできる場合、または AT コマンドをサポートしていない場合、開発者は独自のセルラー抽象化機能を実装してネットワークトランスポートインターフェイスの移植と統合することができます。

MQTT バージョン 3 から coreMQTT への移行

この移行ガイドでは、アプリケーションを MQTT から coreMQTT に移行する方法について説明します。

OTA アプリケーションのバージョン 1 からバージョン 3 への移行

本ガイドは、ご利用のアプリケーションを OTA ライブラリのバージョン 1 からバージョン 3 に移行させる際に役立ちます。

> **Note**
>
> OTA バージョン 2 の API は OTA v3 API と同じであるため、ご利用のアプリケーションで使われている API がバージョン 2 の場合は、API コールを変更する必要はありませんが、ライブラリのバージョン 3 を統合することをお勧めします。

OTA バージョン 3 のデモは、こちらから利用できます。

- ota_demo_core_mqtt
- ota_demo_core_http
- ota_ble

API の変更の概要

OTA ライブラリバージョン 1 とバージョン 3 の API の主な変更点

OTA バージョン 1 の API	OTA バージョン 3 の API	変更点の説明
OTA_AgentInit	OTA_Init	OTA v3 での実装の変更により、入力パラメータに加え、関数から返される値も変更されました。詳細については、下記の OTA_Init のセクションを参照してください。
OTA_AgentShutdown	OTA_Shutdown	MQTT トピックのサブスクライブを解除するオプションの追加パラメータを含む、入力パラメータが変更されまし

OTA バージョン 1 の API	OTA バージョン 3 の API	変更点の説明
		た。詳細については、下記の OTA_Shutdown のセクションを参照してください。
OTA_GetAgentState	OTA_GetState	API 名が変更されましたが、入力パラメータに変更はありません。戻り値は同じですが、列挙型とメンバーの名前が変更されました。詳細については、下記の OTA_GetState のセクションを参照してください。
該当なし	OTA_GetStatistics	OTA_GetPacketsReceived、OTA_GetPacketsQueued、OTA_GetPacketsProcessed、OTA_GetPacketsDropped に代わる、新しく追加された API です。詳細については、下記の OTA_GetStatistics のセクションを参照してください。
OTA_GetPacketsReceived	該当なし	バージョン 3 からは、この API が削除され、OTA_GetStatistics に置き換えられました。
OTA_GetPacketsQueued	該当なし	バージョン 3 からは、この API が削除され、OTA_GetStatistics に置き換えられました。

OTA バージョン 1 の API	OTA バージョン 3 の API	変更点の説明
OTA_GetPacketsProcessed	該当なし	バージョン 3 からは、この API が削除され、OTA_GetStatistics に置き換えられました。
OTA_GetPacketsDropped	該当なし	バージョン 3 からは、この API が削除され、OTA_GetStatistics に置き換えられました。
OTA_ActivateNewImage	OTA_ActivateNewImage	入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。詳細については、OTA_ActivateNewImage のセクションを参照してください。
OTA_SetImageState	OTA_SetImageState	入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、パラメータの名前が変更されました。また、返される OTA エラーコードの名前も変更され、新しいエラーコードが追加されました。詳細については、OTA_SetImageState のセクションを参照してください。

OTA バージョン 1 の API	OTA バージョン 3 の API	変更点の説明
OTA_GetImageState	OTA_GetImageState	入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、列挙型の戻り値の名前が変更されました。詳細については、OTA_GetImageState のセクションを参照してください。
OTA_Suspend	OTA_Suspend	入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。詳細については、OTA_Suspend のセクションを参照してください。
OTA_Resume	OTA_Resume	OTA ライブラリのバージョン 3 では、接続の処理が OTA デモ/アプリケーション内で実行されるため、接続用の入力パラメータは削除されました。また、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。詳細については、OTA_Resume のセクションを参照してください。

OTA バージョン 1 の API	OTA バージョン 3 の API	変更点の説明
OTA_CheckForUpdate	OTA_CheckForUpdate	入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。詳細については、OTA_CheckForUpdate のセクションを参照してください。
該当なし	OTA_EventProcessingTask	新しく追加された API で、メインイベントループによって OTA 更新のイベントが処理されます。呼び出しは、アプリケーションタスクで実行する必要があります。詳細については、OTA_EventProcessingTask のセクションを参照してください。
該当なし	OTA_SignalEvent	新しく追加された API で、発生したイベントは OTA イベントキューの最後尾に追加されます。エージェントタスクを通知する際に、内部 OTA モジュールによって使用される API です。詳細については、OTA_SignalEvent のセクションを参照してください。
該当なし	OTA_Err_strerror	OTA エラーをエラーコードから文字列に変換する、新しい API です。

OTA バージョン 1 の API	OTA バージョン 3 の API	変更点の説明
該当なし	OTA_JobParse_strerror	ジョブの解析エラーをエラーコードから文字列に変換する、新しい API です。
該当なし	OTA_OsStatus_strerror	OTA OS 移植のステータスをエラーコードから文字列に変換する、新しい API です。
該当なし	OTA_PalStatus_strerror	OTA PAL 移植のステータスをエラーコードから文字列に変換する、新しい API です。

必要な変更の説明

OTA_Init

バージョン 1 の OTA エージェントを初期化する際、接続コンテキスト、モノの名前、完全なコールバック、タイムアウトのパラメータを入力として受け取る `OTA_AgentInit` API が使用されます。

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

この API は、OTA、OTA インターフェイス、モノの名前、およびアプリケーションコールバックに必要なバッファのパラメータを有する `OTA_Init` に変更されました。

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

削除された入力パラメータ:

pvConnectionContext:

OTA ライブラリバージョン 3 では、接続コンテキストをパラメータに渡す必要がなく、MQTT/HTTP オペレーションは OTA デモ/アプリケーションの各インターフェイスによって処理されるため、接続コンテキストのパラメータは削除されました。

xTicksToWait:

OTA_Init を呼び出す前に、OTA デモ/アプリケーションでタスクが作成されるため、ティックのパラメータは削除されました。

名前が変更された入力パラメータ:

xFunc:

パラメータの名前が OtaAppCallback に変更され、パラメータタイプは OtaAppCallback_t に変更されました。

新しい入力パラメータ:

pOtaBuffer

初期化時に OtaAppBuffer_t 構造体を使用して、アプリケーションでバッファの割り当てを行い、それらを OTA ライブラリに渡す必要があります。必要となるバッファは、ファイルのダウンロードに使用されるプロトコルによって若干異なります。MQTT プロトコルの場合は、ストリーミング名のバッファが必要であり、HTTP プロトコルの場合は、署名付き URL と認可スキームのバッファが必要です。

ファイルのダウンロードに MQTT を使用する場合に必要となるバッファは、次のとおりです。

```
static OtaAppBuffer_t otaBuffer =
{
    .pUpdateFilePath    = updateFilePath,
    .updateFilePathsize = otaexampleMAX_FILE_PATH_SIZE,
    .pCertFilePath      = certFilePath,
    .certFilePathSize   = otaexampleMAX_FILE_PATH_SIZE,
    .pStreamName        = streamName,
    .streamNameSize     = otaexampleMAX_STREAM_NAME_SIZE,
    .pDecodeMemory      = decodeMem,
    .decodeMemorySize   = ( 1U << otaconfigLOG2_FILE_BLOCK_SIZE ),
    .pFileBitmap        = bitmap,
    .fileBitmapSize     = OTA_MAX_BLOCK_BITMAP_SIZE
```

```
};
```

ファイルのダウンロードに HTTP を使用する場合に必要となるバッファは、次のとおりです。

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

各パラメータの意味は次のとおりです。

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

pOtaInterfaces

OTA_Init の 2 つ目の入力パラメータは、OtaInterfaces_t タイプの OTA インターフェースへの参照です。この一連のインターフェイスは、オペレーティングシステムインターフェイ

ス、MQTT インターフェイス、HTTP インターフェイス、およびプラットフォーム抽象化レイヤーインターフェイス内にある OTA ライブラリおよびインクルードに渡す必要があります。

OTA OS インターフェイス

OTA OS 関数インターフェイスは、デバイスが OTA ライブラリを使用するために実装する必要のある API 群です。このインターフェイスの関数を実装すると、ユーザーアプリケーション内の OTA ライブラリに提供されます。通常はオペレーティングシステムによって提供される機能を実行する際に、OTA ライブラリが関数の実装を呼び出します。例えば、イベント、タイマー、およびメモリ割り当ての管理機能がこれに含まれます。FreeRTOS および POSIX 用の実装は、OTA ライブラリで提供されます。

提供済みの FreeRTOS 移植を使用した FreeRTOS 用の例は以下のとおりです。

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

提供済みの POSIX 移植を使用した Linux 用の例は以下のとおりです。

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

MQTT インターフェイス

OTA MQTT インターフェイスは、OTA ライブラリがストリーミングサービスからファイルブロックをダウンロードできるようにするために、ライブラリに実装する必要がある API 群です。

MQTT を介した OTA のデモの coreMQTT エージェント API の使用例は、次のとおりです。

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

HTTP インターフェイス

OTA HTTP インターフェイスは、OTA ライブラリが署名付き URL に接続してデータブロックを取得することで、ファイルブロックをダウンロードできるようにするために、ライブラリに実装する必要のある API 群です。ストリーミングサービスを使わず、署名付き URL からダウンロードするように OTA ライブラリを設定しない限り、このインターフェイスの実行は任意です。

HTPP を介した OTA のデモの coreHTTP API の使用例は、次のとおりです。

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

OTA PAL インターフェイス

OTA PAL インターフェイスは、デバイスが OTA ライブラリを使用するために実装する必要のある API 群です。OTA PAL のデバイス固有の実装は、ユーザーアプリケーションのライブラリに提供されます。これらの機能は、ダウンロードしたものの保存、管理、および認証をライブラリが行う際に使用されます。

```
OtaInterfaces_t otaInterfaces;
otaInterfaces.pal.getPlatformImageState = otaPal_GetPlatformImageState;
otaInterfaces.pal.setPlatformImageState = otaPal_SetPlatformImageState;
otaInterfaces.pal.writeBlock = otaPal_WriteBlock;
otaInterfaces.pal.activate = otaPal_ActivateNewImage;
```

```
    otaInterfaces.pal.closeFile = otaPal_CloseFile;
    otaInterfaces.pal.reset = otaPal_ResetDevice;
    otaInterfaces.pal.abort = otaPal_Abort;
    otaInterfaces.pal.createFile = otaPal_CreateFileForRx;
```

戻り値の変更点:

戻り値は、OTA エージェントの状態から OTA エラーコードに変更されました。「AWS IoT Over-the-air Update v3.0.0 : OtaErr_t」を参照してください。

OTA_Shutdown

OTA エージェントを停止させる際に使用する API は、OTA ライブラリバージョン 1 では OTA_AgentShutdown でしたが、バージョン 3 からは OTA_Shutdown になり、入力パラメータも変更されました。

OTA エージェントのシャットダウン (バージョン 1)

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

OTA エージェントのシャットダウン (バージョン 3)

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

ticksToWait:

OTA エージェントがシャットダウンプロセスを完了するまでの待機時間をティック数で示します。これをゼロに設定すると、待機時間なしで機能がすぐに戻ります。実際の状態は、呼び出し元に返されます。この間、エージェントはスリープ状態にはならず、ビジーループに使用されます。

新しい入力パラメータ:

unsubscribeFlag:

シャットダウンが呼び出されたときに、ジョブトピックからサブスクライブを解除するオペレーションを実行する必要があるかどうかを示すフラグです。フラグが 0 の場合、ジョブトピックのサブスクライブ解除のオペレーションは呼び出されません。ジョブトピックからアプリケーションをサブスクライブ解除する必要がある場合、OTA_Shutdown を呼び出す際には、このフラグを 1 に設定する必要があります。

戻り値の変更点:

OtaState_t:

OTA エージェントの状態の列挙型とそのメンバーの名前が変更されました。「AWS IoT Over-the-air Update v3.0.0」を参照してください。

OTA_GetState

API 名が OTA_AgentGetState から OTA_GetState に変更されました。

OTA エージェントのシャットダウン (バージョン 1)

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

OTA エージェントのシャットダウン (バージョン 3)

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

戻り値の変更点:

OtaState_t:

OTA エージェントの状態の列挙型とそのメンバーの名前が変更されました。「AWS IoT Over-the-air Update v3.0.0」を参照してください。

OTA_GetStatistics

統計用に新しく追加された、単一の API です。この API は、OTA_GetPacketsReceived、OTA_GetPacketsQueued、OTA_GetPacketsProcessed、OTA_GetPackets に取って代わるものです。また、OTA ライブラリバージョン 3 では、実行中のジョブのみに統計番号が関連付けられます。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

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

pStatistics:

実行中のジョブにおけるパケットの受信、ドロップ、キューへの追加、処理などの統計データ用の入出力パラメータです。

出力パラメータ:

OTA エラーコード。

使用例:

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

OTA_ActivateNewImage

入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

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

返される OTA エラーコードの列挙型が変更され、新しいエラーコードが追加されました。「AWS IoT Over-the-air Update v3.0.0 : OtaErr_t」を参照してください。

使用例:

```
 OtaErr_t otaErr = OtaErrNone;
```

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

OTA_SetImageState

入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、パラメータの名前が変更されました。また、返される OTA エラーコードの名前も変更され、新しいエラーコードが追加されました。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

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

入力パラメータの名前が OtaImageState_t に変更されました。「AWS IoT Over-the-air Update v3.0.0」を参照してください。

返される OTA エラーコードの列挙型が変更され、新しいエラーコードが追加されました。「AWS IoT Over-the-air Update v3.0.0 / OtaErr_t」を参照してください。

使用例:

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

OTA_GetImageState

入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、列挙型の戻り値の名前が変更されました。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

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

列挙型の戻り値の名前は OtaImageState_t に変更されました。「AWS IoT Over-the-air Update v3.0.0 : OtaImageState_t」を参照してください。

使用例:

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

OTA_Suspend

入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

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

返される OTA エラーコードの列挙型が変更され、新しいエラーコードが追加されました。「AWS IoT Over-the-air Update v3.0.0 : OtaErr_t」を参照してください。

使用例:

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

OTA_Resume

OTA ライブラリのバージョン 3 では、接続の処理が OTA デモ/アプリケーション内で実行されるため、接続用の入力パラメータは削除されました。また、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

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

返される OTA エラーコードの列挙型が変更され、新しいエラーコードが追加されました。「AWS IoT Over-the-air Update v3.0.0 : OtaErr_t」を参照してください。

使用例:

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

OTA_CheckForUpdate

入力パラメータは同じですが、OTA ライブラリのバージョン 3 では、返される OTA エラーコードの名前が変更され、新しいエラーコードが追加されました。

OTA ライブラリバージョン 1

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

OTA ライブラリバージョン 3

```
OtaErr_t OTA_CheckForUpdate( void )
```

返される OTA エラーコードの列挙型が変更され、新しいエラーコードが追加されました。「AWS IoT Over-the-air Update v3.0.0 : OtaErr_t」を参照してください。

OTA_EventProcessingTask

新しく追加された API で、メインイベントループによって OTA 更新のイベントが処理されます。呼び出しは、アプリケーションタスクで実行する必要があります。このタスクがアプリケーションに

よって終了されるまで、イベントループは、OTA 更新のために受信したイベントの処理と実行を続けます。

OTA ライブラリバージョン 3

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

FreeRTOS の場合の例は次のとおり。

```
/* Create FreeRTOS task*/
xTaskCreate( prvOTAAgentTask,
             "OTA Agent Task",
             otaexampleAGENT_TASK_STACK_SIZE,
             NULL,
             otaexampleAGENT_TASK_PRIORITY,
             NULL );

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

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

POSIX の場合の例は次のとおり。

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

   /* Handle error. */
}

/* Call OTA_EventProcessingTask from the thread.*/
static void * otaThread( void * pParam )
{
```

```
    /* Calling OTA agent task. */
    OTA_EventProcessingTask( pParam );
    LogInfo( ( "OTA Agent stopped." ) );

    return NULL;
}
```

OTA_SignalEvent

発生したイベントを OTA イベントキューの最後尾に追加する、新しく追加された API です。また、エージェントタスクを通知する際に、内部 OTA モジュールによって使用されます。

OTA ライブラリバージョン 3

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

使用例:

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

OTA ライブラリをサブモジュールとしてご利用のアプリケーションに統合

OTA ライブラリを独自のアプリケーションに統合する場合は、git submodule コマンドを使用します。git submodule コマンドを使用すると、Git リポジトリを別の Git リポジトリのサブディレクトリとして保存できます。OTA ライブラリバージョン 3 は、ota-for-aws-iot-embedded-sdk リポジトリ内に保持されています。

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

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

```
git push
```

詳細については、FreeRTOS ユーザーガイドのアプリケーションへの OTA エージェントの統合を参照してください。

リファレンス

- OTAv1
- OTAv3

OTA PAL 移植のバージョン 1 からバージョン 3 への移行

無線通信経由の更新ライブラリによって、フォルダ構造に加え、ライブラリとデモアプリケーションに必要な設定の配置にいくつかの変更が加えられました。v1.2.0 と連携するよう設計された OTA アプリケーションをライブラリの v3.0.0 に移行させるには、この移行ガイドに従って、PAL 移植機能の署名を更新し、複数の設定ファイルを追加する必要があります。

OTA PAL への変更

- OTA PAL 移植のディレクトリ名は、ota から ota_pal_for_aws に更新されました。このフォルダには、ota_pal.c と ota_pal.h の 2 つのファイルが含まれている必要があります。PAL のヘッダーファイル libraries/freertos_plus/aws/ota/src/aws_iot_ota_pal.h は OTA ライブラリから削除されているため、ポート内で定義する必要があります。
- リターンコード (OTA_Err_t) は、列挙型 OTAMainStatus_t に変換されます。変換後のリターンコードについては、ota_platform_interface.h を参照してください。また、ヘルパーマクロも用意されているので、OtaPalMainStatus と OtaPalSubStatus のコードをマージしたり、OtaMainStatus を OtaPalStatus などから抽出したりする際に活用できます。
- PAL でのログ記録
  - DEFINE_OTA_METHOD_NAME マクロを削除。
  - 以前: `OTA_LOG_L1( "[%s] Receive file created.\r\n", OTA_METHOD_NAME );`
  - 更新後: `LogInfo(( "Receive file created."));` (目的のログを取得する際は、LogDebug、LogWarn、LogError を使用すること)
- 変数 cOTA_JSON_FileSignatureKey を OTA_JsonFileSignatureKey に変更。

関数

関数の署名は ota_pal.h で定義され、プレフィックスは prvPAL ではなく otaPal を使用します。

> **Note**
>
> PAL の正確な名前は、技術的には自由に設定できますが、資格認定テストに対応させるためには、以下に示す名前と一致させる必要があります。

- バージョン 1: `OTA_Err_t prvPAL_CreateFileForRx( OTA_FileContext_t * const *C* );`

  バージョン 3: `OtaPalStatus_t otaPal_CreateFileForRx( OtaFileContext_t * const *pFileContext* );`

  注: まとまった受信データを格納する新しい受信ファイルを作成します。

- バージョン 1: `int16_t prvPAL_WriteBlock( OTA_FileContext_t * const C, uint32_t ulOffset, uint8_t * const pcData, uint32_t ulBlockSize );`

  バージョン 3: `int16_t otaPal_WriteBlock( OtaFileContext_t * const pFileContext, uint32_t ulOffset, uint8_t * const pData, uint32_t ulBlockSize );`

  注: 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。

- バージョン 1: `OTA_Err_t prvPAL_ActivateNewImage( void );`

  バージョン 3: `OtaPalStatus_t otaPal_ActivateNewImage( OtaFileContext_t * const *pFileContext* );`

  注: OTA 経由で受信した最新の MCU イメージをアクティブ化します。

- バージョン 1: `OTA_Err_t prvPAL_ResetDevice( void );`

  バージョン 3: `OtaPalStatus_t otaPal_ResetDevice( OtaFileContext_t * const *pFileContext* );`

  注: デバイスをリセットします。

- バージョン 1: `OTA_Err_t prvPAL_CloseFile( OTA_FileContext_t * const *C* );`

  バージョン 3: `OtaPalStatus_t otaPal_CloseFile( OtaFileContext_t * const *pFileContext* );`

  注: 指定された OTA コンテキストの基盤となっている受信ファイルを認証し、閉じます。

- バージョン 1: `OTA_Err_t prvPAL_Abort( OTA_FileContext_t * const *C* );`

  バージョン 3: `OtaPalStatus_t otaPal_Abort( OtaFileContext_t * const *pFileContext* );`

  注: OTA 転送を停止します。

- バージョン 1: `OTA_Err_t prvPAL_SetPlatformImageState( OTA_ImageState_t *eState* );`

  バージョン 3: `OtaPalStatus_t otaPal_SetPlatformImageState( OtaFileContext_t * const pFileContext, OtaImageState_t eState );`

  注: OTA 更新イメージの状態の設定を実施します。
- バージョン 1: `OTA_PAL_ImageState_t prvPAL_GetPlatformImageState( void );`

  バージョン 3: `OtaPalImageState_t otaPal_GetPlatformImageState( OtaFileContext_t * const *pFileContext* );`

  注: OTA 更新イメージの状態を取得します。

データ型

- バージョン 1: `OTA_PAL_ImageState_t`

  ファイル: `aws_iot_ota_agent.h`

  バージョン 3: `OtaPalImageState_t`

  ファイル: `ota_private.h`

  注: プラットフォームの実装によって設定されたイメージの状態。
- バージョン 1: `OTA_Err_t`

  ファイル: `aws_iot_ota_agent.h`

  バージョン 3: `OtaErr_t OtaPalStatus_t (combination of OtaPalMainStatus_t and OtaPalSubStatus_t)`

  ファイル: `ota.h`、`ota_platform_interface.h`

  注: v1: 符号のない 32 個の整数を定義するマクロ。v3: エラーの種類を表す特殊な列挙型であり、エラーコードに関連付けられた型。
- バージョン 1: `OTA_FileContext_t`

  ファイル: `aws_iot_ota_agent.h`

バージョン 3: OtaFileContext_t

ファイル: ota_private.h

注: v1: データの列挙型とバッファを含む。v3: 追加のデータ長変数を含む。

- バージョン 1: OTA_ImageState_t

  ファイル: aws_iot_ota_agent.h

  バージョン 3: OtaImageState_t

  ファイル: ota_private.h

  注: OTA イメージの状態

設定変更

ファイル aws_ota_agent_config.h の名前が ota_config.h に変更されました。これにより、インクルードガードが _AWS_OTA_AGENT_CONFIG_H_ から OTA_CONFIG_H_ に変更されます。

- ファイル aws_ota_codesigner_certificate.h は、削除されました。
- デバッグメッセージを印刷するための新しいログ記録スタックが追加されました。

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

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

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

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

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

#include "logging_stack.h"

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

- 定数設定が追加されました。

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

新規ファイル: ota_demo_config.h には、コード署名の証明書やアプリケーションバージョンなど、OTA デモで必要な設定が含まれています。

- demos/include/aws_ota_codesigner_certificate.h で定義された signingcredentialSIGNING_CERTIFICATE_PEM は、otapalconfigCODE_SIGNING_CERTIFICATE として ota_demo_config.h に移動しました。なお、次のように指定すると PAL ファイルからアクセスできます。

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

  ファイル aws_ota_codesigner_certificate.h は、削除されました。
- APP_VERSION_BUILD、APP_VERSION_MINOR、APP_VERSION_MAJOR のマクロが ota_demo_config.h に追加されました。バージョン情報を含む tests/include/aws_application_version.h、libraries/c_sdk/standard/common/include/iot_appversion32.h、demos/demo_runner/aws_demo_version.c などの古いファイルは、削除されました。

OTA PAL テストの変更点

- 「Full_OTA_AGENT」テストグループとそのすべての関連ファイルを削除しました。従来、このテストグループは資格に必要でした。これらのテストは OTA ライブラリ用であり、OTA PAL 移植に特有のものではありません。現在の OTA ライブラリは、OTA リポジトリでホストされるテストをすべて網羅しているため、このテストグループは不要となりました。

- 「Full_OTA_CBOR」と「Quarantine_OTA_CBOR」テストグループを削除し、そのすべての関連ファイルも削除しました。これらのテストは、資格認定テストに含まれていませんでした。削除されたテストの対象だった機能については、OTA リポジトリでテストされています。
- テストファイルをライブラリディレクトリから tests/integration_tests/ota_pal ディレクトリに移動させました。
- OTA ライブラリ API の v3.0.0 を使用するよう、OTA PAL 資格認定テストを更新しました。
- OTA PAL テストが、テスト用のコード署名の証明書にアクセスする方法を更新しました。以前は、コード署名の認証情報専用のヘッダーファイルが存在しましたが、新しいバージョンのライブラリは、その点が変更されています。テストコードでは、この変数が ota_pal.c で定義されることを想定しています。その値は、プラットフォーム固有の OTA 設定ファイルで定義されたマクロに割り当てられます。

チェックリスト

このチェックリストを使用して、移行に必要な手順を実行していることを確認します。

- OTA PAL 移植フォルダの名前を ota から ota_pal_for_aws に更新します。
- 上記の機能を含んだファイル ota_pal.h を追加します。ota_pal.h ファイルの例については、GitHub を参照してください。
- 設定ファイルを追加します。
  - ファイル名を aws_ota_agent_config.h から ota_config.h に変更するか、新しく という名前のファイルを作成します。
    - 追加:

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

    - インクルード:

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

  - 上記のファイルを aws_test config フォルダにコピーし、ota_demo_config.h に含まれているものをすべて aws_test_ota_config.h に置き換えます。
  - ota_demo_config.h ファイルを追加します。
  - aws_test_ota_config.h ファイルを追加します。
- ota_pal.c に以下の変更を加えます。

  - インクルードを最新の OTA ライブラリファイル名に更新します。
  - DEFINE_OTA_METHOD_NAME マクロを削除します。
  - OTA PAL 関数の署名を更新します。
  - ファイルコンテキスト変数の名前を C から pFileContext に更新します。
  - OTA_FileContext_t 構造体とそれに関連するすべての変数を更新します。
  - cOTA_JSON_FileSignatureKey を OTA_JsonFileSignatureKey に更新します。
  - OTA_PAL_ImageState_t と Ota_ImageState_t のタイプを更新します。
  - エラータイプとエラー値を更新します。
  - 印刷マクロを更新し、ログ記録スタックを使用するようにします。
  - otapalconfigCODE_SIGNING_CERTIFICATE になるよう signingcredentialSIGNING_CERTIFICATE_PEM を更新します。
  - otaPal_CheckFileSignature と otaPal_ReadAndAssumeCertificate の関数のコメントを更新します。
- CMakeLists.txt ファイルを更新します。
- IDE プロジェクトを更新します。

ドキュメント履歴

次の表は、「FreeRTOS 移植ガイド」と「FreeRTOS 認定ガイド」のドキュメント履歴をまとめたものです。

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2022 年 5 月	FreeRTOS 移植ガイド FreeRTOS 認定ガイド	• FreeRTOS 長期サポート (LTS) ライブラリに基づいて、既存のテストの更新、新しいテストの追加、冗長テストの除去が行われました。詳細については、GitHub の FreeRTOS ライブラリ統合テストの「202205.00」を参照してください。 • 更新済み FreeRTOS 移植フローチャート。 • 新たに「ネットワークトランスポートインターフェイスの移植」が追加されました。 • 認定に AWS IoT over-the-air (OTA) 更新ライブラリの移植が必要になりました。	202012.04-LTS 202112.00

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
		• Wi-Fi と TLS 抽象化移植ガイドは不要になったため削除されました。 • FreeRTOS 認定に関するその他の最新情報については、「Latest changes」を参照してください。	
2021 年 7 月	202107.00 (移植ガイド) 202107.00 (資格ガイド)	• リリース 202107.00 • AWS IoT over-the-air (OTA) 更新ライブラリの移植 を変更 • 「OTA アプリケーションのバージョン 1 からバージョン 3 への移行」を追加 • 「OTA PAL 移植のバージョン 1 からバージョン 3 への移行」を追加	202107.00

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2020 年 12 月	202012.00 (移植ガイド) 202012.00 (資格ガイド)	• リリース 202012.00 • 「coreHTTP ライブラリの設定」を追加 • 「セルラーインターフェイスライブラリの移植」を追加	202012.00
2020 年 11 月	202011.00 (移植ガイド) 202011.00 (資格ガイド)	• リリース 202011.00 • 「coreMQTT ライブラリの設定」を追加	202011.00
2020 年 7 月	202007.00 (移植ガイド) 202007.00 (資格ガイド)	• リリース 202007.00	202007.00
2020 年 2 月 18 日	202002.00 (移植ガイド) 202002.00 (認定ガイド)	• リリース 202002.00 • Amazon FreeRTOS を FreeRTOS に変更	202002.00
2019 年 12 月 17 日	201912.00 (移植ガイド) 201912.00 (認定ガイド)	• リリース 201912.00 • 共通 I/O ライブラリの移植が追加されました。	201912.00

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2019 年 10 月 29 日	201910.00 (移植ガイド) 201910.00 (認定ガイド)	• リリース 201910.00 • 乱数生成器移植の最新情報。	201910.00
2019 年 8 月 26 日	201908.00 (移植ガイド) 201908.00 (認定ガイド)	• リリース 201908.00 • テスト用の HTTPS クライアントライブラリの設定を追加 corePKCS11 ライブラリの移植 の更新	201908.00
2019 年 6 月 17 日	201906.00 (移植ガイド) 201906.00 (資格ガイド)	• リリース 201906.00 • 更新されたディレクトリ構造化	201906.00 メジャー
2019 年 5 月 21 日	1.4.8 (移植ガイド) 1.4.8 (認定ガイド)	• 移植ドキュメントを FreeRTOS 移植ガイドに移動 • 資格ドキュメントを FreeRTOS 資格ガイドに移動	1.4.8
2019 年 2 月 25 日	1.1.6	• 入門ガイドのテンプレートの付録からダウンロードと設定手順を削除 (84 ページ)	1.4.5 1.4.6 1.4.7

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2018 年 12 月 27 日	1.1.5	• 資格チェックリストの付録を CMake 要件により更新 (70 ページ)	1.4.5 1.4.6
2018 年 12 月 12 日	1.1.4	• TCP/IP の移植の付録に lwIP 移植手順を追加 (31 ページ)	1.4.5
2018 年 11 月 26 日	1.1.3	• Bluetooth Low Energy 移植の付録を追加 (52 ページ) • ドキュメント全体に AWS IoT Device Tester for FreeRTOS テスト情報を追加 • FreeRTOS コンソールの付録に、出品に関する情報への CMake リンクを追加 (85 ページ)	1.4.4

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2018 年 11 月 7 日	1.1.2	• PKCS#11 移植の付録で PKCS#11 PAL インターフェイスの移植手順を更新 (38 ページ) • `CertificateConfigurator.html` へのパスを更新 (76 ページ) • 入門ガイドのテンプレートの付録を更新 (80 ページ)	1.4.3

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2018 年 10 月 8 日	1.1.1	• 新しい「Required for AFQP」列を`aws_test_runner_config.h` テスト設定テーブルに追加 (16 ページ) • 「Create the Test Project」セクションの Unity モジュールディレクトリパスを更新 (14 ページ) • 「Recommended Porting Order」グラフを更新 (22 ページ) • TLS 付録、Test Setup のクライアント証明書とキー変数名を更新 (40 ページ) • Secure Sockets の移植の付録、Test Setup (34 ページ)、TLS の移植の付録、Test Setup (40 ページ)、および TLS Server Setup の付録 (57 ページ) のファイルパスを変更	1.4.2

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2018 年 8 月 27 日	1.1.0	• OTA 更新の移植の付録を追加 (47 ページ) • ブートローダー移植の付録を追加 (51 ページ)	1.4.0 1.4.1
2018 年 8 月 9 日	1.0.1	• 「Recommended Porting Order」グラフを更新 (22 ページ) • PKCS#11 の移植の付録を更新 (36 ページ) • TLS の移植の付録、Test Setup (40 ページ)、および TLS Server Setup の付録、ステップ 9 (51 ページ) のファイルパスを変更 • MQTT の移植の付録、前提条件のハイパーリンクを修正 (45 ページ) • BYOC を作成する手順の付録 (57 ページ) の例に AWS CLI 設定手順を追加しました。	1.3.1 1.3.2

日付	ドキュメントバージョン	変更履歴	FreeRTOS バージョン
2018 年 7 月 31 日	1.0.0	FreeRTOS 資格認定プログラムガイドの初期バージョン	1.3.0

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。