aws

移植指南

免费 RTOS

免费 RTOS: 移植指南

Table of Contents

FreeRTOS 移植

什么是 FreeRTOS

与世界领先的芯片公司合作开发了 20 年，现在每 170 秒有一次下载，FreeRTOS 是面向微控制器和小型微处理器的市场领先的实时操作系统 (RTOS)。根据 MIT 开源许可免费分发，FreeRTOS 包含一个内核和一组持续增加的库，可广泛应用于各个行业领域。FreeRTOS 的设计非常注重可靠性和易用性。FreeRTOS 包含用于连接、安全性和空中下载 (OTA) 更新的库，还包含一些在取得资格的主板上演示 FreeRTOS 特征的演示应用程序。

有关更多信息，请访问 FreeRTOS.org。

将 FreeRTOS 移植到您的 IoT 主板

您需要根据其功能和应用将 FreeRTOS 软件库移植到基于微控制器的主板上。

将 FreeRTOS 移植到您的设备

1. 按照 下载要移植的 FreeRTOS 中的说明，下载要移植的 FreeRTOS 的最新版本。
2. 按照 为移植设置工作区和项目 中的说明，配置所下载的 FreeRTOS 中的文件和文件夹以进行移植和测试。
3. 按照 移植 FreeRTOS 库 中的说明，将 FreeRTOS 库移植到您的设备。每个移植主题包含有关测试移植的说明。

移植常见问题

什么是 FreeRTOS 移植？

> FreeRTOS 移植是指特定于主板的、所需 FreeRTOS 库和您的平台支持的 FreeRTOS 内核的 API 实现。通过移植，API 可在主板上工作，并实现设备驱动程序与平台供应商提供的 BSP 之间的必需集成。您的移植还应包含主板所需的所有配置调整（如时钟频率，堆栈大小、堆大小）。

如果您对移植有疑问，但在本页或《FreeRTOS 移植指南》的其余部分未得到解答，请查看可用的 FreeRTOS 支持选项。

下载要移植的 FreeRTOS

从 freertos.org 下载最新的 FreeRTOS 或长期支持 (LTS) 版本，或者从 GitHub (FreeRTOS-LTS) 或 (FreeRTOS) 克隆。

> **Note**
>
> 我们建议您克隆存储库。通过进行克隆，在更新被推送到存储库时，您可以更轻松地获得主分支的更新。

或者，也可以为 FreeRTOS 或 FreerTOS-LTS 存储库中的各个库创建子模块。但是，请确保库版本与 FreeRTOS 或 FreerTOS-LTS 存储库中 `manifest.yml` 文件中列出的组合相匹配。

下载或克隆 FreeRTOS 后，您可以开始将 FreeRTOS 库移植到您的主板上。有关说明，请参阅为移植设置工作区和项目，然后参阅移植 FreeRTOS 库。

为移植设置工作区和项目

请按照以下步骤设置工作区和项目：

- 使用所选项目结构和构建系统来导入 FreeRTOS 库。
- 使用开发主板支持的集成式开发环境 (IDE) 和工具链创建项目。
- 在项目中包含主板支持包 (BSP) 和主板专用驱动程序。

工作区设置完成后，就可以开始移植单个 FreeRTOS 库了。

移植 FreeRTOS 库

开始移植之前，按照为移植设置工作区和项目中的说明进行操作。

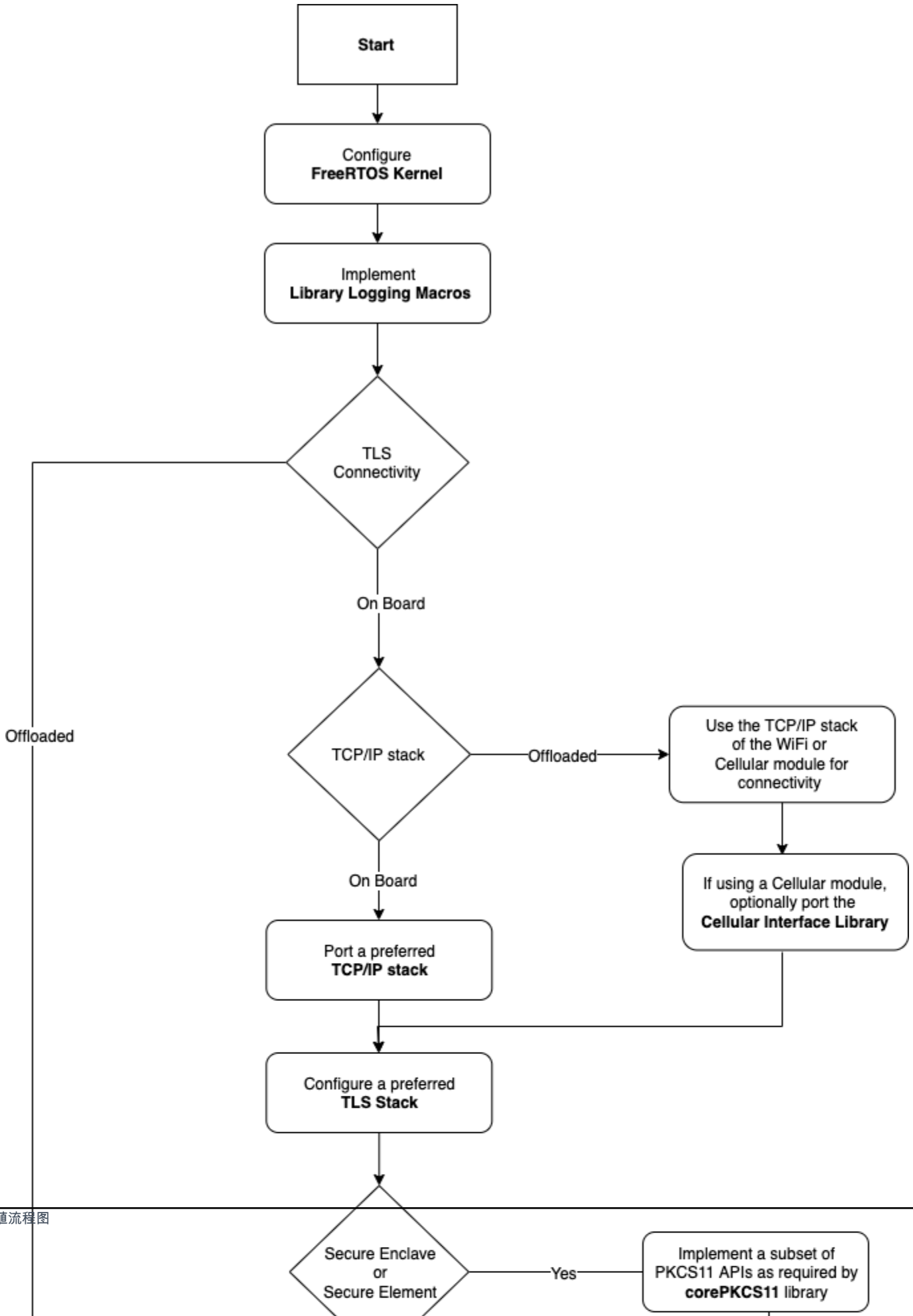
FreeRTOS 移植流程图 描述了移植所需的库。

要将 FreeRTOS 移植到您的设备，请按照以下主题中的说明操作。

1. 配置 FreeRTOS 内核移植
2. 实现库日志记录宏
3. 移植 TCP/IP 堆栈
4. 移植网络传输接口
5. 移植 corePKCS11 库
6. 配置 coreMQTT 库
7. 配置 coreHTTP 库
8. 移植 AWS IoT over-the-air (OTA) 更新库
9. 移植蜂窝接口库

FreeRTOS 移植流程图

在将 FreeRTOS 移植到主板时，请使用下面的移植流程图作为可视化辅助工具。


Start
Configure
FreeRTOS Kernel
Implement
Library Logging Macros
TLS
Connectivity
On Board
Offloaded
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

配置 FreeRTOS 内核移植

此部分提供有关将 FreeRTOS 内核移植集成到 FreeRTOS 移植测试项目的说明。有关可用内核移植的列表，请参阅 FreeRTOS 内核移植。

FreeRTOS 使用 FreeRTOS 内核来支持多任务处理和任务间通信。有关更多信息，请参阅《FreeRTOS 用户指南》和 FreerTOS.org 中的 FreeRTOS 内核基础知识和 FreeRTOS.org。

> **Note**
>
> 将 FreeRTOS 内核移植到新架构不在本文档的讨论范围内。如果您有兴趣，请联系 FreeRTOS 工程团队。
>
> 对于 FreeRTOS 资格认证计划，仅支持现有 FreeRTOS 内核端口。该计划不接受对这些移植进行修改。有关更多信息，请查看 FreeRTOS 内核移植策略。

先决条件

要设置 FreeRTOS 内核以进行移植，需要具备以下条件：

- 用于目标平台的正式 FreeRTOS 内核移植或 FreeRTOS 支持的移植。
- 一个 IDE 项目，其中包括用于目标平台和编译器的正确 FreeRTOS 内核端口文件。有关设置测试项目的信息，请参阅为移植设置工作区和项目。

配置 FreeRTOS 内核

FreeRTOS 内核使用名为 `FreeRTOSConfig.h` 的配置文件进行自定义。该文件为内核指定应用程序特定的配置设置。有关每个配置选项的说明，请参阅 FreeRTOS.org 上的自定义。

要将 FreeRTOS 内核配置为与您的设备配合使用，请包含 `FreeRTOSConfig.h` 并修改任何其他 FreeRTOS 配置。

有关每个配置选项的说明，请参阅 FreeRTOS.org 上的自定义配置。

测试

- 运行一个简单的 FreeRTOS 任务，将消息记录到串行输出控制台。

- 验证消息是否按预期输出到控制台。

实现库日志记录宏

FreeRTOS 库使用以下日志记录宏，这些宏按详细程度的递增顺序列出。

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

必须为所有宏提供定义。建议是：

- 宏应支持 C89 样式的记录记录。
- 日志记录应具备线程安全性。来自多个任务的日志行不得相互交错。
- 日志记录 API 不得阻止，并且必须使应用程序任务免于在 I/O 上被阻止。

有关实现规范，请参阅 FreerTos.org 上的日志记录功能。您可以在此示例中查看一种实现。

测试

- 运行包含多个任务的测试，以便验证日志不会交错。
- 运行测试以验证日志 API 不会在 I/O 上被阻止。
- 使用各种标准（例如，C89,C99 样式日志记录）测试日志记录宏。
- 通过设置不同的日志级别（例如，Debug、Info、Error 和 Warning）来测试日志记录宏。

移植 TCP/IP 堆栈

本节提供移植和测试板载 TCP/IP 堆栈的说明。如果您的平台将 TCP/IP 和 TLS 功能分载到单独的网络处理器或模块，您可以跳过此移植部分并访问移植网络传输接口。

FreeRTOS+TCP 是 FreeRTOS 内核的原生 TCP/IP 堆栈。FreeRTOS+TCP 由 FreeRTOS 工程团队维护，是推荐与 FreeRTOS 配合使用的 TCP/IP 堆栈。有关更多信息，请参阅移植 FreeRTOS+TCP。或者，您可以使用第三方 TCP/IP 堆栈 lwIP。本节提供的测试说明使用适用于 TCP 纯文本的传输接口测试，并且不依赖于具体实现的 TCP/IP 堆栈。

移植 FreeRTOS+TCP

FreeRTOS+TCP 是 FreeRTOS 内核的原生 TCP/IP 堆栈。有关更多信息，请参阅 FreeRTOS.org。

先决条件

要移植 FreeRTOS+TCP 库，您需要以下信息：

- 包括供应商提供的以太网或 Wi-Fi 驱动程序在内的 IDE 项目。

  有关设置测试项目的信息，请参阅为移植设置工作区和项目。
- FreeRTOS 内核的经验证配置。

  有关为您的平台配置 FreeRTOS 内核的信息，请参阅配置 FreeRTOS 内核移植。

移植

在开始移植 FreeRTOS+TCP 库之前，请检查 GitHub 目录以了解是否已存在您的主板的移植。

如果移植不存在，请执行以下操作：

1. 按照 FreeRTOS.org 上将 FreeRTOS+TCP 移植到不同微控制器的说明，将 FreeRTOS+TCP 移植到您的设备。
2. 如有必要，请按照 FreeRTOS.org 上将 FreeRTOS+TCP 移植到新的嵌入式 C 编译器的说明，将 FreeRTOS+TCP 移植到新的编译器。
3. 在名为 `NetworkInterface.c` 的文件中实施使用供应商提供的以太网或 Wi-Fi 驱动程序的新移植。访问 GitHub 存储库以获取模板。

在创建移植后，或者如果移植已存在，请创建 `FreeRTOSIPConfig.h` 并编辑配置选项，以使它们适合您的平台。有关配置选项的更多信息，请参阅 FreeRTOS.org 上的 FreeRTOS+TCP 配置。

测试

无论您使用的是 freerTOS+TCP 库还是第三方库，请按照以下步骤进行测试：

- 在传输接口测试中提供 `connect/disconnect/send/receive` API 的实现。
- 在纯文本 TCP 连接模式下设置 Echo 服务器，然后运行传输接口测试。

> **Note**
>
> 要使设备正式获得 FreeRTOS 的资格，如果您的架构需要移植 TCP/IP 软件堆栈，则需要在 AWS IoT Device Tester 的纯文本 TCP 连接模式下根据传输接口测试验证设备移植的源代码。按照《FreeRTOS 用户指南》中使用适用于 FreeRTOS 的 AWS IoT Device Tester中的说明为移植验证设置 AWS IoT Device Tester。要测试特定库的移植，必须在 Device Tester `configs` 文件夹下面的 `device.json` 文件中启用正确的测试组。

移植 corePKCS11 库

公有密钥加密标准 #11 定义了一个独立于平台的 API，可用于管理和使用加密令牌。PKCS 11 是指它定义的标准和 API。PKCS #11 加密 API 用于提取密钥存储、加密对象的 get/set 属性以及会话语义。该标准在操纵常见加密对象时应用广泛。它的功能允许应用程序软件使用、创建、修改和删除加密对象，而无需将这些对象暴露在应用程序的内存中。

FreeRTOS 库和参考集成使用 PCKS #11 接口标准的子集，重点是涉及非对称密钥、随机数生成和哈希的操作。下表列出了使用案例和需要支持的 PKCS #11 API。

使用案例

使用案例	必需的 PKCS #11 API 系列
全部	初始化、最终确定、打开/关闭会话、GetsLotList、登录
预置	GenerateKeyPair、CreateObject、DestroyObject、InitToken、GetTokenInfo
TLS	随机、签名、FindObject、GetAttributeValue
FreeRTOS+TCP	随机，摘要
OTA	验证、摘要、FindObject、GetAttributeValue

何时实现完整的 PKCS #11 模块

在通用闪存中存储私有密钥对于评估和快速原型设计场景非常方便。在生产场景中，为了减少数据窃取和设备复制的威胁，我们建议您使用专用加密硬件。加密硬件包含具有防止导出加密密钥功能的组件。为了支持这一点，您必须实现使用上表中定义的 FreeRTOS 库所需的 PKCS #11 子集。

何时使用 FreeRTOS corePKCS11

corePKCS11 库包含 PKCS #11 接口 (API) 的基于软件的实现，该接口使用 Mbed TLS 提供的加密功能。这是在没有专用加密硬件的情况下，为硬件的快速原型设计和评估场景提供的。在这种情况下，您只需要实现 corePKCS11 PAL 即可让基于 corePKCS11 软件的实现与您的硬件平台配合使用。

移植 corePKCS11

您必须有实现才能读取加密对象并将其写入非易失性存储器 (NVM)，例如板载闪存。加密对象必须存储在设备重新编程时未初始化且未擦除的 NVM 部分。corePKCS11 库的用户将使用凭证预置设备，然后使用可通过 corePKCS11 接口访问这些凭证的新应用程序重新编程设备。corePKCS11 PAL 移植必须提供一个位置来存储：

- 设备客户端证书
- 设备客户端私有密钥
- 设备客户端公有密钥
- 受信任的根 CA
- 代码验证公有密钥（或包含代码验证公有密钥的证书），用于安全引导加载程序和空中下载 (OTA) 更新
- 即时预置证书。

包含头文件并实现定义的 PAL API。

PAL API

函数	描述
PKCS11_PAL_Initialize	初始化 PAL 层。由 corePKCS11 库在其初始化序列开始时调用。
PKCS11_PAL_SaveObject	将数据写入非易失性存储。

函数	描述
PKCS11_PAL_FindObject	使用 PKCS #11 CKA_LABEL 来在非易失性存储中搜索相应的 PKCS #11 对象，并返回该对象的句柄（如果存在）。
PKCS11_PAL_GetObjectValue	检索对象的值，给定句柄。
PKCS11_PAL_GetObjectValueCleanup	PKCS11_PAL_GetObjectValue 调用的清除。可用于释放 PKCS11_PAL_GetObjectValue 调用中分配的内存。

测试

如果您使用 FreeRTOS corePKCS11 库或实现所需的 PKCS11 API 子集，则必须通过 FreeRTOS PKCS11 测试。这些测试用于验证 FreeRTOS 库所需的函数能否按预期执行。

本节还介绍了如何使用资格认证测试在本地运行 FreeRTOS PKCS11 测试。

先决条件

要设置 FreeRTOS PKCS11 测试，必须实现以下几点。

- PKCS11 API 支持的移植。
- FreeRTOS 资格认证测试平台功能的实现，其中包括：
  - FRTest_ThreadCreate
  - FRTest_ThreadTimedJoin
  - FRTest_MemoryAlloc
  - FRTest_MemoryFree

（参阅 GitHub 上有关 PKCS #11 的 FreeRTOS 库集成测试的 README.md 文件。）

移植测试

- 将 FreeRTOS-Libraries-Integration-Tests 作为子模块添加到您的项目中。只要可以构建子模块，就可以将其放在项目的任何目录中。

- 将 config_template/test_execution_config_template.h 和 config_template/test_param_config_template.h 复制到构建路径中的项目位置，然后将其重命名为test_execution_config.h和test_param_config.h。
- 将相关文件包含到构建系统中。如果使用 CMake，则可以使用 qualification_test.cmake 和 src/pkcs11_tests.cmake 来包含相关文件。
- 实现 UNITY_OUTPUT_CHAR，这样，测试输出日志就不会与设备日志交错。
- 集成 MbedTLS，用于验证 cryptoki 的操作结果。
- 从应用程序调用 RunQualificationTest()。

配置测试

PKCS11 测试套件必须根据 PKCS11 实现进行配置。下表在 test_param_config.h 头文件中列出了 PKCS11 测试所需的配置。

PKSC11 测试配置

配置	描述
PKCS11_TEST_RSA_KEY_SUPPORT	该移植支持 RSA 密钥功能。
PKCS11_TEST_EC_KEY_SUPPORT	该移植支持 EC 密钥功能。
PKCS11_TEST_IMPORT_PRIVATE_KEY_SUPPORT	移植支持导入私有密钥。如果已启用支持密钥功能，则将在测试中验证 RSA 和 EC 密钥导入。
PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT	移植支持生成密钥对。如果已启用支持密钥功能，则将在测试中验证 EC 密钥对生成。
PKCS11_TEST_PREPROVISIONED_SUPPORT	移植具有预置凭证。PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS 、PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS 和 PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS 是证书的示例。
PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS	测试中使用的私有密钥的标签。

配置	描述
PKCS11_TEST_LABEL_DEVICE_PUBLIC_KEY_FOR_TLS	测试中使用的公有密钥的标签。
PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS	测试中使用的证书的标签。
PKCS11_TEST_JITP_CODEVERIFY_ROOT_CERT_SUPPORTED	该移植支持 JITP 的存储。将其设置为 1 可启用 JITP `codeverify` 测试。
PKCS11_TEST_LABEL_CODE_VERIFICATION_KEY	JITP `codeverify` 测试中使用的代码验证密钥的标签。
PKCS11_TEST_LABEL_JITP_CERTIFICATION	JITP `codeverify` 测试中使用的 JITP 证书的标签。
PKCS11_TEST_LABEL_ROOT_CERTIFICATE	JITP `codeverify` 测试中使用的根证书的标签。

FreeRTOS 库和参考集成必须支持至少一种密钥功能配置，例如 RSA 或 Elliptic 曲线密钥，以及 PKCS11 API 支持的一种密钥预置机制。该测试必须启用以下配置：

- 至少启用以下密钥功能配置之一：
  - PKCS11_TEST_RSA_KEY_SUPPORT
  - PKCS11_TEST_EC_KEY_SUPPORT
- 至少启用以下密钥预置配置之一：
  - PKCS11_TEST_IMPORT_PRIVATE_KEY_SUPPORT
  - PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT
  - PKCS11_TEST_PREPROVISIONED_SUPPORT

预先配置的设备凭证测试必须在以下条件下运行：

- 必须启用 `PKCS11_TEST_PREPROVISIONED_SUPPORT` 并禁用其他配置机制。
- 只有一个密钥功能（`PKCS11_TEST_RSA_KEY_SUPPORT` 或 `PKCS11_TEST_EC_KEY_SUPPORT`）处于启用状态。

- 根据您的密钥功能设置预先配置的密钥标签，包括 PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS、PKCS11_TEST_LABEL_DEVICE_PUBLIC_K 和 PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS。在运行测试之前，这些凭证必须存在。

如果实现支持预先配置的凭证和其他配置机制，则测试可能需要使用不同的配置运行多次。

> **Note**
>
> 如果启用 PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT 或 PKCS11_TEST_GENERATE_KEYPAIR_SUPPORT，则会在测试期间销毁带有标签 PKCS11_TEST_LABEL_DEVICE_PRIVATE_KEY_FOR_TLS、PKCS11_TEST_LABEL_DEVICE_PUBLI 和 PKCS11_TEST_LABEL_DEVICE_CERTIFICATE_FOR_TLS 的对象。

运行测试

本节介绍如何通过资格认证测试在本地测试 PKCS11 接口。或者，您也可以使用 IDT 自动执行。有关详细信息，请参阅《FreeRTOS 用户指南》中 适用于 FreeRTOS 的 AWS IoT Device Tester。

以下说明介绍了如何运行测试：

- 打开 test_execution_config.h 并将 CORE_PKCS11_TEST_ENABLED 定义为 1。
- 构建应用程序并将其刷写到您的设备上，以便运行。测试结果会输出到串行端口。

下面是输出测试结果的一个示例。

```
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
```

当所有测试均已通过后，测试完成。

> **Note**
>
> 要正式对设备进行 FreeRTOS 资格认证，您必须使用 AWS IoT Device Tester 验证设备的移植源代码。按照《FreeRTOS 用户指南》中使用适用于 FreeRTOS 的 AWS IoT Device Tester中的说明为移植验证设置 AWS IoT Device Tester。要测试特定库的移植，必须在 AWS IoT Device Tester `configs` 文件夹下面的 `device.json` 文件中启用正确的测试组。

移植网络传输接口

集成 TLS 库

传输层安全性协议 (TLS) 身份验证，请使用首选 TLS 堆栈。我们建议使用 Mbed TLS，因为它已通过 FreeRTOS 库进行了测试。您可以在此 GitHub 存储库中找到此类示例。

无论您的设备使用哪种 TLS 实现，都必须为 TCP/IP 堆栈的 TLS 堆栈实现底层传输钩子。它们必须支持 AWS IoT 支持的 TLS 密码套件。

移植网络传输接口库

您必须实现一个网络传输接口才能使用 coreMQTT 和 coreHTTP。网络传输接口包含在单个网络连接上发送和接收数据所需的函数指针和上下文数据。有关更多详细信息，请参阅传输接口。FreeRTOS 提供了一组内置的网络传输接口测试来验证这些实现。以下部分将指导您设置项目以运行这些测试。

先决条件

要完成此测试，您需要具备以下条件：

- 一个带有构建系统的项目，该系统可以使用经过验证的 FreeRTOS 内核移植来构建 FreeRTOS。
- 网络驱动程序的工作实现。

移植

- 将 FreeRTOS-Libraries-Integration-Tests 作为子模块添加到您的项目中。只要子模块可以构建，将其放在项目中的哪个位置并不重要。
- 将 `config_template/test_execution_config_template.h` 和 `config_template/test_param_config_template.h` 复制到构建路径中的项目位置，然后将其重命名为`test_execution_config.h`和`test_param_config.h`。
- 将相关文件包含到构建系统中。如果使用 CMake，则使用 `qualification_test.cmake` 和 `src/transport_interface_tests.cmake` 来包含相关文件。
- 在适当的项目位置实现以下功能：
- • `network connect function`：签名由 `src/common/network_connection.h` 中的 `NetworkConnectFunc` 定义。此函数接收指向网络上下文的指针、指向主机信息的指针和指向网络凭证的指针。它使用提供的网络凭证与主机信息中指定的服务器建立连接。
  - `network disconnect function`：签名由 `src/common/network_connection.h` 中的 `NetworkDisconnectFunc` 定义。此函数接收指向网络上下文的指针。它断开存储在网络上下文中先前建立的连接。
  - `setupTransportInterfaceTestParam()`：在 `src/transport_interface/transport_interface_tests.h` 中定义。实现的名称和签名必须与 `transport_interface_tests.h` 定义的完全相同。此函数接收指向 TransportInterfaceTestParam 结构的指针。它将填充传输接口测试使用的 TransportInterfaceTestParam 结构中的字段。
- 实现 UNITY_OUTPUT_CHAR，这样，测试输出日志就不会与设备日志交错。

- 从应用程序调用 `runQualificationTest()`。在调用之前，必须正确初始化设备硬件并连接网络。

凭证管理（设备端生成的密钥）

将 `test_param_config.h` 中的 FORCE_GENERATE_NEW_KEY_PAIR 设置为 1 时，设备应用程序会生成新的设备端密钥对并输出公有密钥。在与 Echo 服务器建立 TLS 连接时，设备应用程序使用 ECHO_SERVER_ROOT_CA 和 TRANSPORT_CLIENT_CERTIFICATE 作为 Echo 服务器根 CA 和客户端证书。IDT 在资格认证运行期间设置这些参数。

凭据管理（导入密钥）

在与 Echo 服务器建立 TLS 连接时，设备应用程序使用 `test_param_config.h` 中的 ECHO_SERVER_ROOT_CA、TRANSPORT_CLIENT_CERTIFICATE 和 TRANSPORT_CLIENT_PRIVATE_KEY 作为 Echo 服务器根 CA、客户端证书和客户端私有密钥。IDT 在资格认证运行期间设置这些参数。

测试

本节介绍如何通过资格认证测试在本地测试传输接口。有关更多详细信息，可在 GitHub 上的 FreerTOS-Libraries-Integration-Tests 的 transport_interface 部分提供的 README.md 文件中找到。

或者，您也可以使用 IDT 自动执行。有关详细信息，请参阅《FreeRTOS 用户指南》中 适用于 FreeRTOS 的 AWS IoT Device Tester。

启用测试

打开 `test_execution_config.h` 并将 TRANSPORT_INTERFACE_TEST_ENABLED 定义为 1。

设置测试的 Echo 服务器

本地测试需要一台可从运行测试的设备访问的 Echo 服务器。如果传输接口实现支持 TLS，则 Echo 服务器必须支持 TLS。如果您还没有，FreeRTOS-Libraries-Integration-Tests GitHub 存储库提供了 Echo 服务器实现。

配置测试的项目

在 `test_param_config.h` 中，将 ECHO_SERVER_ENDPOINT 和 ECHO_SERVER_PORT 更新为上一步中的端点和服务器设置。

设置凭证（设备端生成的密钥）

- 将 ECHO_SERVER_ROOT_CA 设置为 Echo 服务器的服务器证书。
- 将 FORCE_GENERATE_NEW_KEY_PAIR 设置为 1 以生成密钥对并获取公有密钥。
- 生成密钥后，将 FORCE_GENERATE_NEW_KEY_PAIR 设置回 0。
- 使用公有密钥和服务器密钥以及证书生成客户端证书。
- 将 TRANSPORT_CLIENT_CERTIFICATE 设置为生成的客户端证书。

安装凭证（导入密钥）

- 将 ECHO_SERVER_ROOT_CA 设置为 Echo 服务器的服务器证书。
- 将 TRANSPORT_CLIENT_CERTIFICATE 设置为预先生成的客户端证书。
- 将 TRANSPORT_CLIENT_PRIVATE_KEY 设置为预先生成的客户端私有密钥。

构建并刷写应用程序

使用您选择的工具链构建和刷写应用程序。调用 `runQualificationTest()` 时，将运行传输接口测试。测试结果会输出到串行端口。

> **Note**
>
> 要使设备正式获得 FreeRTOS 的资格，您必须使用 AWS IoT Device Tester 为 OTA PAL 和 OTA E2E 测试组验证设备的移植源代码。按照《FreeRTOS 用户指南》中使用适用于 FreeRTOS 的 AWS IoT Device Tester中的说明为移植验证设置 AWS IoT Device Tester。要测试特定库的移植，必须在 AWS IoT Device Tester `configs` 文件夹下面的 `device.json` 文件中启用正确的测试组。

配置 coreMQTT 库

边缘设备可以使用 MQTT 协议与 AWS 云通信。AWS IoT 中托管一个 MQTT 代理，可与连接的边缘设备相互发送和接收消息。

coreMQTT 库会为运行 FreeRTOS 的设备实现 MQTT 协议。不需要移植 coreMQTT 库，但您设备的测试项目必须通过所有 MQTT 测试才能获得资格。有关更多信息，请参阅《FreeRTOS 用户指南》中的 coreMQTT 库。

先决条件

要设置 coreMQTT 库测试，您需要一个网络传输接口移植。要了解更多信息，请参阅移植网络传输接口。

测试

运行 coreMQTT 集成测试：

- 向 MQTT 代理注册客户端证书。
- 在 `config` 中设置代理端点并运行集成测试。

创建参考 MQTT 演示

我们建议使用 coreMQTT 代理来处理所有 MQTT 操作的线程安全。用户还需要发布和订阅任务以及 Device Advisor 测试，以验证应用程序是否有效地集成了 TLS、MQTT 和其他 FreeRTOS 库。

要使设备正式获得 FreeRTOS 的资格，请使用 AWS IoT Device Tester MQTT 测试用例验证您的集成项目。有关设置和测试的说明，请参阅 AWS IoT Device Advisor 工作流程。下面列出了 TLS 和 MQTT 的强制测试用例：

TLS 测试用例

测试用例	测试用例	必需测试
TLS	TLS 连接	是
TLS	TLS 支持 AWS IoT 密码套件	推荐的密码套件
TLS	TLS 不安全服务器证书	是
TLS	TLS 主题名称服务器证书不正确	是

MQTT 测试用例

测试用例	测试用例	必需测试
MQTT	MQTT Connect	是

测试用例	测试用例	必需测试
MQTT	MQTT Connect 抖动重试次数	是，无警告
MQTT	MQTT 订阅	是
MQTT	MQTT 发布	是
MQTT	MQTT ClientPuback QoS1	是
MQTT	MQTT No Ack PingResp	是

配置 coreHTTP 库

边缘设备可以使用 HTTP 协议来与 AWS 云通信。AWS IoT 服务中托管了一个 HTTP 服务器，可与连接的边缘设备相互发送和接收消息。

测试

请按照以下步骤进行测试：

- 设置 PKI 以使用 AWS 或 HTTP 服务器进行 TLS 双向身份验证。
- 运行 CoreHTTP 集成测试。

移植 AWS IoT over-the-air (OTA) 更新库

通过 FreeRT over-the-air OS (OTA) 更新，您可以执行以下操作：

- 将固件映像部署到单个设备、一组设备或整个机群。
- 在将设备添加到组，或重置或重新预配置设备时，将固件部署到设备。
- 将新固件部署到设备之后，验证其真实性和完整性。
- 监控部署进度。
- 调试失败的部署。
- 使用代码签名对固件进行数字签名。 AWS IoT

有关更多信息，请参阅 FreeRTOS 用户指南中的 FreeRTOS 无线更新以及 O 更新文档。AWS IoT ver-the-air

您可以使用 OTA 更新库将 OTA 功能集成到您的 FreeRTOS 应用程序中。有关更多信息，请参阅《FreeRTOS 用户指南》中的 FreeRTOS OTA 更新库。

FreeRTOS 设备必须在它们接收到的 OTA 固件映像上强制实施加密代码签名验证。我们建议采用下列算法：

- 椭圆曲线数字签名算法 (ECDSA)
- NIST P256 曲线
- SHA-256 哈希

先决条件

- 完成为移植设置工作区和项目中的说明。
- 创建网络传输接口端口。

  有关信息，请参阅 移植网络传输接口。
- 集成 coreMQTT 库。请参阅《FreeRTOS 用户指南》中的 coreMQTT 库。
- 创建 OTA 更新的引导加载程序。

平台移植

您必须提供 OTA 便携式抽象层 (PAL) 的实现才能将 OTA 库移植到新设备。PAL API 在 ota_platform_interface.h 文件中定义，必须提供具体的实现详细信息。

函数名称	描述
`otaPal_Abort`	停止 OTA 更新。
`otaPal_CreateFileForRx`	创建一个文件来存储收到的数据块。
`otaPal_CloseFile`	关闭指定的文件。如果使用实现加密保护的存储，则可能会对文件进行身份验证。
`otaPal_WriteBlock`	按照指定偏移量将数据块写入指定文件。如果成功，则返回写入的字节数。否则，该函数返回负

函数名称	描述
	错误代码。数据块大小始终为二的乘方，并且将保持对齐。有关更多信息，请参阅 OTA 库配置。
otaPal_ActivateNewImage	激活或启动新的固件映像。对于某些端口，如果以编程方式同步重置设备，此函数将不会返回。
otaPal_SetPlatformImageState	执行平台所需的任何操作来接受或拒绝最新的 OTA 固件映像（或包）。要实现此函数，请参阅您的主板（平台）的文档以了解详细信息和架构。
otaPal_GetPlatformImageState	获取 OTA 更新映像的状态。

如果您的设备已内置相应支持，实施此表中的函数。

函数名称	描述
otaPal_CheckFileSignature	验证指定文件的签名。
otaPal_ReadAndAssumeCertificate	从文件系统读取指定的签署人证书并将其返回给调用方。
otaPal_ResetDevice	重置设备。

> **Note**
>
> 确保您有支持 OTA 更新的引导加载程序。有关创建 AWS IoT 设备引导加载程序的说明，请参阅IoT 设备引导加载程序。

E2E 和 PAL 测试

运行 OTA PAL 和 E2E 测试。

E2E 测试

OTA 端到端 (E2E) 测试用于验证设备的 OTA 能力和模拟现实场景。该测试包括错误处理。

先决条件

要完成此测试，您需要具备以下条件：

- 一个集成了 AWS OTA 库的项目。有关更多信息，请访问《OTA 库移植指南》。
- 使用 OTA 库移植演示应用程序与 AWS IoT Core 交互以进行 OTA 更新。请参阅 移植 OTA 演示应用程序。
- 设置 IDT 工具。这将运行 OTA E2E 主机应用程序来构建、刷写和监控具有不同配置的设备，并验证 OTA 库的集成。

移植 OTA 演示应用程序

OTA E2E 测试必须有 OTA 演示应用程序才能验证 OTA 库的集成。该演示应用程序必须具有执行 OTA 固件更新的能力。你可以在 FreeRTOS 存储库中找到 FreeRTOS OTA 演示应用程序。 GitHub我们建议您使用该演示应用程序作为参考，并根据您的规范进行修改。

移植测试

1. 初始化 OTA 代理。
2. 实现 OTA 应用程序回调函数。
3. 创建 OTA 代理事件处理任务。
4. 启动 OTA 代理。
5. 监控 OTA 代理统计信息。
6. 关闭 OTA 代理。

有关详细说明，请访问 FreeRTOS OTA over MQTT - 演示的入口点。

配置

必须使用以下配置才能与之交互 AWS IoT Core：

- AWS IoT Core 客户凭证
  - 在 `Ota_Over_Mqtt_Demo/demo_config.h` 中使用 Amazon Trust Services 端点设置 democonfigROOT_CA_PEM。有关更多详细信息，请参阅 AWS 服务器身份验证。

  - 使用你的客户端凭据设置 democonfigClient_certificate_pem 和 democonfigClient_Private_key_pem。 `Ota_Over_Mqtt_Demo/demo_config.h` AWS IoT 要了解有关客户端证书和私有密钥的信息，请参阅 AWS 客户端身份验证详细信息。
- 应用程序版本
- OTA 控制协议
- OTA 数据协议
- 代码签名凭证
- 其他 OTA 库配置

在 FreeRTOS OTA 演示应用程序中，您可以在 `demo_config.h` 和 `ota_config.h` 中找到上述信息。有关更多信息，请访问 FreeRTOS OTA over MQTT - 设置设备。

构建验证

运行演示应用程序以运行 OTA 作业。成功完成后，您可以继续运行 OTA E2E 测试。

F reeRTOS OTA 演示提供了有关在 FreeRTOS Windows 模拟器上设置 OTA 客户端和 AWS IoT Core OTA 作业的详细信息。 AWS OTA 同时支持 MQTT 和 HTTP 协议。有关更多详细信息，请参阅以下示例：

- Windows 模拟器上的 OTA over MQTT 演示
- Windows 模拟器上的 OTA over HTTP 演示

使用 IDT 工具运行测试

要运行 OTA 端到端测试，必须使用 AWS IoT Device Tester (IDT) 自动执行。有关更多详细信息，请参阅《FreeRTOS 用户指南》中 适用于 FreeRTOS 的AWS IoT Device Tester。

E2E 测试用例

测试用例	描述
`OTAE2EGreaterVersion`	定期 OTA 更新的满意路径 (Happy Path) 测试。它使用设备成功更新的新版本创建更新。
`OTAE2EBackToBackDownloads`	此测试会连续创建 3 个 OTA 更新。预计设备将连续更新 3 次。

测试用例	描述
OTAE2ERollbackIfUnableToConnectAfterUpdate	此测试会验证当设备无法使用新固件连接到网络时，是否会回滚到以前的固件。
OTAE2ESameVersion	此测试确认，如果传入的固件版本保持不变，则设备会拒绝接收该固件。
OTAE2EUnsignedImage	此测试验证当镜像没有签名时，设备是否会拒绝更新。
OTAE2EUntrustedCertificate	如果固件使用不受信任的证书签名，此测试将验证设备是否会拒绝更新。
OTAE2EPreviousVersion	此测试验证设备是否拒绝了旧更新版本。
OTAE2EIncorrectSigningAlgorithm	不同的设备支持不同的签名和哈希算法。如果设备使用不支持的算法创建，则此测试会验证 OTA 更新是否失败。
OTAE2EDisconnectResume	这是暂停和恢复功能的成功路径测试。此测试创建 OTA 更新并开始更新。然后，它 AWS IoT Core 使用相同的客户端 ID（事物名称）和凭据进行连接。 AWS IoT Core 然后断开设备的连接。设备应检测到已与其断开连接 AWS IoT Core，一段时间后，设备会自行进入暂停状态，然后尝试重新连接 AWS IoT Core 并恢复下载。
OTAE2EDisconnectCancelUpdate	当设备处于暂停状态时，如果取消 OTA 任务，此测试会检查设备能否自行恢复。它的作用与OTAE2EDisconnectResume 测试相同，只是在连接到设备后 AWS IoT Core，它会断开设备连接，从而取消OTA更新。新更新已创建。设备应重新连接到 AWS IoT Core，中止当前更新，返回等待状态，接受并完成下一次更新。

测试用例	描述
OTAE2EPresignedUrlExpired	创建 OTA 更新时，您可以配置 S3 预签名 URL 的生命周期。此测试会验证设备是否能够执行 OTA，即使它无法在 URL 过期时完成下载。此设备应请求新的任务文档，其中包含用于恢复下载的新 URL。
OTAE2E2UpdatesCancel1st	此测试会连续创建两个 OTA 更新。当设备报告正在下载第一个更新时，测试人员会强制取消第一个更新。设备预计会中止当前更新并进行第二次更新，然后完成更新。
OTAE2ECancelThenUpdate	此测试会连续创建两个 OTA 更新。当设备报告正在下载第一个更新时，测试人员会强制取消第一个更新。设备预计会中止当前更新并进行第二次更新，然后完成更新。
OTAE2EImageCrashed	此测试会检查设备是否能够在映像崩溃时拒绝更新。

PAL 测试

先决条件

要移植网络传输接口测试，您需要具备以下几点：

- 一个可使用有效的 FreeRTOS 内核端口构建 FreeRTOS 的项目。
- OTA PAL 的有效实现。

移植

- 将 FreeRTOS-Libraries-Integration-Tests 作为子模块添加到您的项目中。子模块必须位于项目中可构建该子模块的位置。
- 将 `config_template/test_execution_config_template.h` 和 `config_template/test_param_config_template.h` 复制到构建路径中的位置，然后将其重命名为 `test_execution_config.h` 和 `test_param_config.h`。

- 将相关文件包含到构建系统中。如果使用 CMake，则可以使用 qualification_test.cmake 和 src/ota_pal_tests.cmake 来包含相关文件。
- 通过实现以下功能来配置测试：
  - SetupOtaPalTestParam()：在 src/ota/ota_pal_test.h 中定义。实现的名称和签名必须与 ota_pal_test.h 中定义的完全相同。目前，您不需要配置此函数。
- 实现 UNITY_OUTPUT_CHAR，这样，测试输出日志就不会与设备日志交错。
- 从应用程序调用 RunQualificationTest()。在调用之前，必须正确初始化设备硬件并连接网络。

测试

本节介绍 OTA PAL 资格认证测试的本地测试。

启用测试

打开 test_execution_config.h 并将 OTA_PAL_TEST_ENABLED 定义为 1。

在 test_param_config.h 中，更新以下选项：

- OTA_PAL_TEST_CERT_TYPE：选择使用的证书类型。
- OTA_PAL_CERTIFICATE_FILE：设备证书的路径（如果适用）。
- OTA_PAL_FIRMWARE_FILE：固件文件的名称（如果适用）。
- OTA_PAL_USE_FILE_SYSTEM：如果 OTA PAL 使用文件系统抽象，则设置为 1。

使用您选择的工具链构建和刷写应用程序。调用 RunQualificationTest() 时，OTA PAL 测试将运行。测试结果会输出到串行端口。

集成 OTA 任务

- 在您当前的 MQTT 演示中添加 OTA 代理。
- 使用运行 OTA 端到端 (E2E) 测试。 AWS IoT这会验证集成是否按预期运行。

> **Note**
>
> 要使设备正式获得FreeRTOS的资格，您必须使用OTA PAL和OTA E2E测试组验证设备的移植源代码。 AWS IoT Device Tester按照《FreeRTOS 用户指南》中的 “用 AWS IoT Device

Tester 于 FreeRTOS” 中的说明进行端口验证设置。 AWS IoT Device Tester 要测试特定库的端口，必须在 AWS IoT Device Tester `configs`文件夹`device.json`的文件中启用正确的测试组。

IoT 设备引导加载程序

您必须提供自己的安全引导加载程序应用程序。确保设计和实施能够适当缓解安全威胁。以下是供您参考的威胁建模。

IoT 设备引导加载程序的威胁建模

背景

作为一个可行的定义，此威胁模型所指的嵌入式 AWS IoT 设备是基于微控制器的与云服务交互的产品。它们可以部署在消费者、商业或工业环境中。IoT 设备可以收集有关用户、患者、机器或环境的数据，并且可以控制从灯泡和门锁到工厂机器的任何事物。

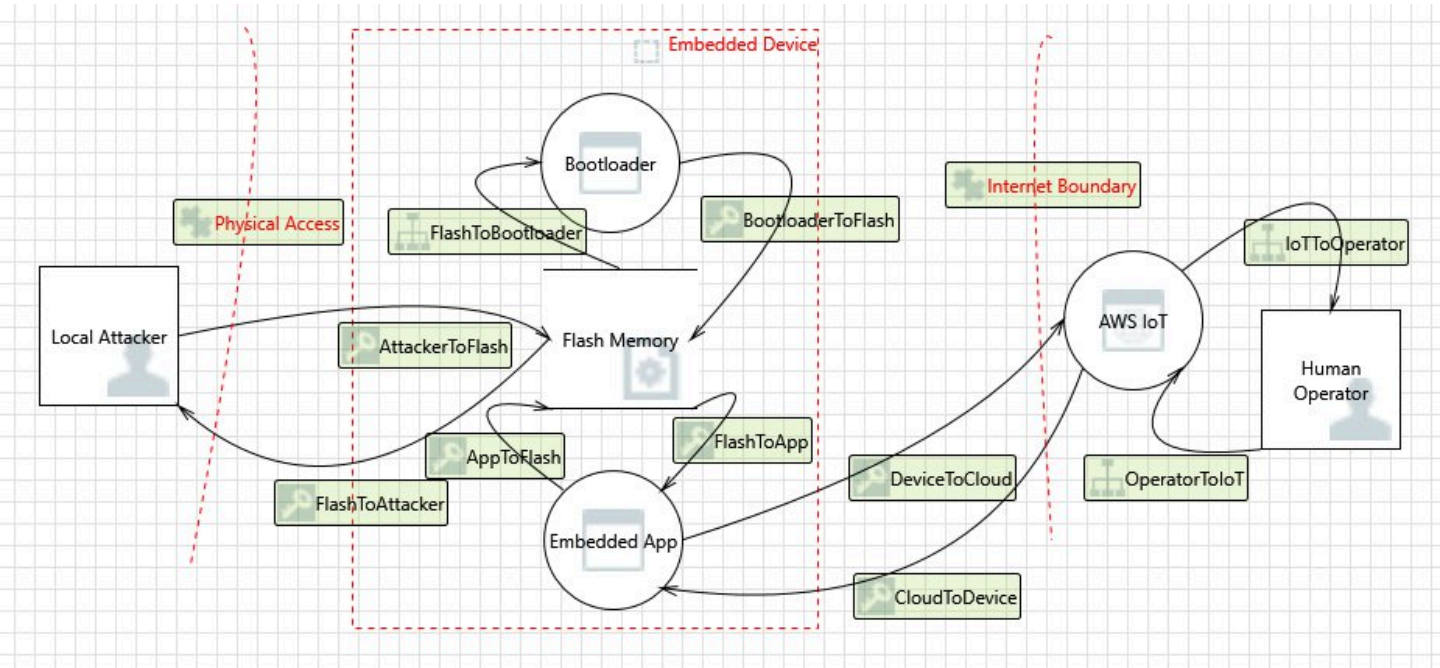
从假设对手的角度看，威胁建模是一种保护安全的方法。通过考虑对手的目标和方法，创建了一个威胁列表。威胁是对手对资源或资产发起的攻击。该列表具有优先级，并用于确定或创建缓解解决方案。在选择缓解解决方案时，实施和维护这些解决方案的成本应与它提供的实际安全价值平衡。可以使用多种威胁模型方法。每种产品都能够支持安全且成功的 AWS IoT 产品的开发。

FreeRTOS 为设备提供 OTA over-the-air () 软件更新。 AWS IoT 更新工具将云服务与设备上的软件库和合作伙伴提供的引导加载程序结合使用。该威胁模型明确侧重于针对引导加载程序的威胁。

引导加载程序使用案例

- 在部署之前对固件进行数字签名和加密。
- 将新的固件映像部署到单个设备、一组设备或整个队列中。
- 在新固件部署到设备之后，验证其真实性和完整性。
- 设备仅运行来自受信任的来源的未修改软件。
- 设备可以灵活处理通过 OTA 收到的故障软件。

数据流程图



威胁

有些攻击有多种缓解模式；例如，通过验证对 TLS 服务器提供的证书和新固件映像的代码签名者证书的信任度，可以缓解 man-in-the-middle 旨在提供恶意固件映像的网络。为了最大限度提高引导加载程序的安全性，任何非引导加载程序缓解解决方案都被视为不可靠。引导加载程序应具有每种攻击的内置缓解解决方案。采用分层缓解解决方案被称为 defense-in-depth。

威胁：

- 攻击者劫持设备到服务器的连接以提供恶意固件映像。

  缓解措施示例

  - 在引导时，引导加载程序使用已知证书验证映像的加密签名。如果验证失败，引导加载程序将回滚到上一个映像。
- 攻击者利用缓冲区溢出以将恶意行为引入到闪存中存储的现有固件映像。

  缓解措施示例

  - 在引导时，引导加载程序将按照上面所述进行验证。如果验证失败并且没有以前的映像，引导加载程序将停止。
  - 在引导时，引导加载程序将按照上面所述进行验证。如果验证失败且没有以前的映像，引导加载程序将进入仅 OTA 故障保护模式。
- 攻击者将设备引导到以前存储的映像，从而可以利用该映像发起攻击。

缓解措施示例

- 在成功安装和测试新映像后，将擦除存储上一个映像的闪存扇区。
- 在每次成功升级时，都会熔断保险丝，每个映像将会重新熔断保险丝才能运行，除非已熔断正确数量的保险丝。

- OTA 更新提供故障或恶意映像，从而导致设备无法正常运行。

  缓解措施示例

  - 引导加载程序启动硬件监视程序计时器，以触发回滚到上一个映像。
- 攻击者修补引导加载程序以绕过映像验证，因此，设备将接受未签名的映像。

  缓解措施示例

  - 引导加载程序位于 ROM（只读存储器）中，无法进行修改。
  - 引导加载程序位于 OTP（one-time-programmable 内存）中，无法修改。
  - 引导加载程序位于 ARM 的安全区域中 TrustZone，无法修改。
- 攻击者替换验证证书，因此，设备将接受恶意映像。

  缓解措施示例

  - 证书位于加密协处理器中，无法进行修改。
  - 证书位于 ROM、OTP 或安全区域中，无法进行修改。

进一步的威胁建模

该威胁模型仅考虑引导加载程序。进一步的威胁建模可能会提高整体安全性。建议的方法是列出对手的目标、这些目标针对的资产以及资产的入口点。可以考虑为控制资产而对入口点发起的攻击以创建威胁列表。以下是 IoT 设备的目标、资产和入口点示例列表。这些列表并非详尽无遗，旨在为您提供一些启发。

对手的目标

- 敲诈钱财
- 诋毁声誉
- 伪造数据
- 转移资源
- 远程监视目标

- 实际访问站点
- 造成严重破坏
- 灌输恐怖

主要资产

- 私有密钥
- 客户端证书
- CA 根证书
- 安全凭证和令牌
- 客户的个人身份信息
- 商业机密实施
- 传感器数据
- 云分析数据存储
- 云基础设施

入口点

- DHCP 响应
- DNS 响应
- 基于 TLS 的 MQTT
- HTTPS 响应
- OTA 软件映像
- 应用程序指示的其他内容，例如 USB
- 实际访问总线
- 开封 IC

移植蜂窝接口库

FreeRTOS 支持 TCP 分载的蜂窝抽象层的 AT 命令。有关更多信息，请参阅 freertos.org 上的蜂窝接口库和移植蜂窝接口库。

先决条件

蜂窝接口库没有直接依赖关系。但是，在 FreeRTOS 网络堆栈中，以太网、Wi-Fi 和蜂窝无法共存，因此，开发人员必须选择其中之一来与 移植网络传输接口 集成。

> **Note**
>
> 如果蜂窝模块能够支持 TLS 分载，或者不支持 AT 命令，则开发人员可以实现自己的蜂窝抽象来与 移植网络传输接口 集成。

从 MQTT 版本 3 迁移到 coreMQTT

本迁移指南阐述了如何将应用程序从 MQTT 迁移到 coreMQTT。

为 OTA 应用程序从版本 1 迁移到版本 3

本指南将帮助您将应用程序从 OTA 库版本 1 迁移到版本 3。

> **Note**
>
> OTA 版本 2 的 API 与 OTA v3 API 相同，因此，如果您的应用程序使用的是版本 2 的 API，则无需对 API 调用进行更改，但我们建议您集成该库的版本 3。

从下面可以获得 OTA 版本 3 的演示：

- ota_demo_core_mqtt。
- ota_demo_core_http。
- ota_ble。

API 更改摘要

OTA 库版本 1 和版本 3 之间的 API 更改摘要

OTA 版本 1 API	OTA 版本 3 API	更改说明
ota_agentinit	ota_init	由于 OTA v3 中实施的更改，输入参数以及从函数返回的值都发生了变化。有关详细信息，请参阅下面的 OTA_Init 部分。
OTA_AgentShutdown	OTA_Shutdown	对输入参数的更改，包括用于可选取消订阅 MQTT 主题的附加参数。有关详细信息，请参阅下面的 OTA_Shutdown 部分。
OTA_GetAgentState	OTA_GetState	已更改 API 名称，但未更改输入参数。返回值相同，但重命名了枚举和成员。有关详细信

OTA 版本 1 API	OTA 版本 3 API	更改说明
		息，请参阅下面的 OTA_GetState 部分。
不适用	OTA_GetStatistics	添加了新的 API 来取代 OTA_GetPacketsReceived、OTA_GetPacketsQueued、OTA_GetPacketsProcessed、OTA_GetPacketsDropped。有关详细信息，请参阅下面的 OTA_GetStatistics 部分。
OTA_GetPacketsReceived	不适用	已从版本 3 中移除此 API，将其替换成了 OTA_GetStatistics。
OTA_GetPacketsQueued	不适用	已从版本 3 中移除此 API，将其替换成了 OTA_GetStatistics。
OTA_GetPacketsProcessed	不适用	已从版本 3 中移除此 API，将其替换成了 OTA_GetStatistics。
OTA_GetPacketsDropped	不适用	已从版本 3 中移除此 API，将其替换成了 OTA_GetStatistics。
OTA_ActivateNewImage	OTA_ActivateNewImage	输入参数相同，但重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。有关详细信息，请参阅 OTA_ActivateNewImage 部分。

OTA 版本 1 API	OTA 版本 3 API	更改说明
OTA_SetImageState	OTA_SetImageState	输入参数相同但但已重命名，同时重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。有关详细信息，请参阅 OTA_SetImageState 部分。
OTA_GetImageState	OTA_GetImageState	输入参数相同，但在 OTA 库的版本 3 中重命名了返回枚举。有关详细信息，请参阅 OTA_GetImageState 部分。
OTA_Suspend	OTA_Suspend	输入参数相同，但重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。有关详细信息，请参阅 OTA_Suspend 部分。
OTA_Resume	OTA_Resume	移除了连接的输入参数，因为该连接在 OTA 演示/应用程序中进行处理，重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。有关详细信息，请参阅 OTA_Resume 部分。
OTA_CheckForUpdate	OTA_CheckForUpdate	输入参数相同，但重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。有关详细信息，请参阅 OTA_CheckForUpdate 部分。

OTA 版本 1 API	OTA 版本 3 API	更改说明
不适用	OTA_EventProcessingTask	添加了新的 API，它是处理 OTA 更新事件的主事件循环，必须由应用程序任务调用。有关详细信息，请参阅 OTA_EventProcessingTask 部分。
不适用	OTA_SignalEvent	添加了新的 API，它将事件添加到 OTA 事件队列后，并由内部 OTA 模块用来向代理任务发出信号。有关详细信息，请参阅 OTA_SignalEvent 部分。
不适用	OTA_Err_strerror	用于将 OTA 错误的错误代码转换为字符串的新 API。
不适用	OTA_JobParse_strerror	用于将作业解析错误的错误代码转换为字符串的新 API。
不适用	OTA_OsStatus_strerror	用于将 OTA OS 端口状态的状态代码转换为字符串的新 API。
不适用	OTA_PalStatus_strerror	用于将 OTA PAL 端口状态的状态代码转换为字符串的新 API。

所需更改的描述

ota_init

在 v1 中初始化 OTA 代理时会使用 `OTA_AgentInit` API，它将连接上下文、事物名称、完整回调和超时的参数作为输入。

```
OTA_State_t OTA_AgentInit( void * pvConnectionContext,
                           const uint8_t * pucThingName,
```

```
                            pxOTACompleteCallback_t xFunc,
                            TickType_t xTicksToWait );
```

此 API 现已更改为 `OTA_Init`，带有 ota、ota 接口、事物名称和应用程序回调所需的缓冲区的参数。

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

移除了输入参数 -

pvConnectionContext -

移除了连接上下文，因为 OTA 库版本 3 不需要将连接上下文传递给它，而且 MQTT/HTTP 操作由 OTA 演示/应用程序中的相应接口处理。

xTicksToWait -

还移除了等待刻度数参数，因为任务是在调用 OTA_Init 之前在 OTA 演示/应用程序中创建的。

重命名了输入参数 -

xFunc -

该参数已重命名为 OtaAppCallback，并且其类型已更改为 OtaAppCallback_t。

新的输入参数 -

pOtaBuffer

在初始化期间，应用程序必须使用 OtaAppBuffer_t 结构分配缓冲区并将其传递给 OTA 库。根据下载文件所使用的协议，所需的缓冲区略有不同。对于 MQTT 协议，需要流名称的缓冲区，对于 HTTP 协议，则需要预签名 url 和授权方案的缓冲区。

使用 MQTT 下载文件时需要的缓冲区 -

```
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
};
```

使用 HTTP 下载文件时需要的缓冲区 -

```
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
};
```

其中 -

```
    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 的第二个输入参数是对 OtaInterfaces_t 类型的 OTA 接口的引用。必须将这一组接口传递给 OTA 库，并在操作系统接口中包含 MQTT 接口、HTTP 接口和平台抽象层接口。

OTA 操作系统接口

OTA 操作系统功能接口是一组 API，必须为设备实现这些接口才能使用 OTA 库。此接口的函数实现会在用户应用程序中提供给 OTA 库。OTA 库调用函数实现来执行通常由操作系统提供的功能。其中包括管理事件、计时器和内存分配。FreeRTOS 和 POSIX 的实现在 OTA 库中提供。

使用提供的 FreeRTOS 移植的 FreeRTOS 示例 -

```
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;
```

使用提供的 POSIX 端口的 Linux 示例 -

```
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 接口

OTA MQTT 接口是一组 API，必须在库中实现这些接口才能让 OTA 库从流式服务下载文件块。

使用 OTA over MQTT 演示演示中的 coreMQTT 代理 API 的示例 -

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

HTTP 接口

OTA HTTP 接口是一组 API，必须在库中实现这些接口才能让 OTA 库通过连接到预签名 URL 并获取数据块来下载文件块。除非您将 OTA 库配置为从预签名 URL（而不是流式服务）下载，否则它是可选的。

使用 OTA over HTTP 演示中的 coreHTTP API 的示例 -

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

OTA PAL 接口

OTA PAL 接口是一组 API，必须为设备实现这些接口才能使用 OTA 库。OTA PAL 的设备特定实现在用户应用程序中提供给库。库使用这些函数来存储、管理和验证下载。

```
  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;
```

返回结果的变化 -

返回值从 OTA 代理状态更改为了 OTA 错误代码。请参阅 AWS IoT空中下载更新 v3.0.0：OtaErr_t。

OTA_Shutdown

在 OTA 库版本 1 中，用于关闭 OTA 代理的 API 是 OTA_AgentShutdown，由于输入参数的更改，该 API 现在已更改为 OTA_Shutdown。

OTA 代理关闭（版本 1）

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

OTA 代理关闭（版本 3）

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

ticksToWait -

等待 OTA 代理完成关闭过程的刻度数量。如果将其设置为零，该函数将立即返回，无需等待。实际状态将返回给调用方。代理暂时不会进入睡眠状态，而会用于繁忙的循环。

新的输入参数 -

unsubscribeFlag -

用于指示在调用关闭时是否应从作业主题执行取消订阅操作的标记。如果该标记为 0，则不会为作业主题调用取消订阅操作。如果应用程序必须取消订阅作业主题，则在调用 OTA_Shutdown 时必须将此标记设置为 1。

返回结果的变化 -

OtaState_t -

重命名了 OTA 代理状态的枚举及其成员。请参阅 AWS IoT 空中下载更新 v3.0.0。

OTA_GetState

API 名称从 OTA_AgentGetState 更改为了 OTA_GetState。

OTA 代理关闭（版本 1）

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

OTA 代理关闭（版本 3）

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

返回结果的变化 -

OtaState_t -

重命名了 OTA 代理状态的枚举及其成员。请参阅 AWS IoT 空中下载更新 v3.0.0。

OTA_GetStatistics

为统计数据添加了新的单一 API。它取代了 API OTA_GetPacketsReceived、OTA_GetPacketsQueued、OTA_GetPacketsProcessed、OTA_GetPacketsDrop
此外，在 OTA 库版本 3 中，统计数字仅与当前作业相关。

OTA 库版本 1

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

OT3 库版本 3

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

pStatistics -

统计数据的输入/输出参数，例如，为当前作业接收、丢弃、排队和处理的数据包。

输出参数 -

OTA 错误代码。

示例用法 -

```
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

输入参数相同，但重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。

OTA 库版本 1

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

OT3 库版本 3

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

更改了返回的 OTA 错误代码枚举，并添加了新的错误代码。请参阅 AWS IoT空中下载更新 v3.0.0：OtaErr_t。

示例用法 -

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

OTA_SetImageState

输入参数相同但但已重命名，同时重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。

OTA 库版本 1

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

OT3 库版本 3

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

输入参数已重命名为 OtaImageState_t。请参阅 AWS IoT 空中下载更新 v3.0.0。

更改了返回的 OTA 错误代码枚举，并添加了新的错误代码。请参阅 AWS IoT 空中下载更新 v3.0.0 / OtaErr_t。

示例用法 -

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

OTA_GetImageState

输入参数相同，但在 OTA 库的版本 3 中重命名了返回枚举。

OTA 库版本 1

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

OT3 库版本 3

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

返回枚举已重命名为 OtaImageState_t。请参阅 AWS IoT 空中下载更新 v3.0.0：OtaImageState_t。

示例用法 -

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

OTA_Suspend

输入参数相同，但重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。

OTA 库版本 1

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

OT3 库版本 3

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

更改了返回的 OTA 错误代码枚举，并添加了新的错误代码。请参阅 AWS IoT空中下载更新 v3.0.0：OtaErr_t。

示例用法 -

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

OTA_Resume

移除了连接的输入参数，因为该连接在 OTA 演示/应用程序中进行处理，重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。

OTA 库版本 1

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

OT3 库版本 3

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

更改了返回的 OTA 错误代码枚举，并添加了新的错误代码。请参阅 AWS IoT空中下载更新 v3.0.0：OtaErr_t。

示例用法 -

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

OTA_CheckForUpdate

输入参数相同，但重命名了返回的 OTA 错误代码，并在 OTA 库的版本 3 中添加了新的错误代码。

OTA 库版本 1

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

OT3 库版本 3

```
OtaErr_t OTA_CheckForUpdate( void )
```

更改了返回的 OTA 错误代码枚举，并添加了新的错误代码。请参阅 AWS IoT空中下载更新 v3.0.0：OtaErr_t。

OTA_EventProcessingTask

这是一个新的 API，也是处理 OTA 更新事件的主事件循环。它必须由应用程序任务调用。此循环将继续处理和执行接收到的 OTA 更新事件，直到应用程序终止此任务。

OT3 库版本 3

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

FreeRTOS 的示例 -

```
/* 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 的示例 -

```
/* 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

这是一个新的 API，可将事件添加到事件队列后，并由内部 OTA 模块来向单个代理任务发出信号。

OT3 库版本 3

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

示例用法 -

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

将 OTA 库作为子模块集成到应用程序中

如果您希望将 OTA 库集成到自己的应用程序中，则可以使用 git submodule 命令。Git 子模块允许您将一个 Git 仓库作为另一个 Git 存储库的子目录。OTA 库版本 3 在 ota-for-aws-iot-embedded-sdk 存储库中进行维护。

```
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
```

有关更多信息，请参阅《FreeRTOS 用户指南》中的将 OTA 代理集成到应用程序中。

参考

- OTAv1。
- OTAv3。

为 OTA PAL 移植从版本 1 迁移到版本 3

空中下载更新库对文件夹结构以及库和演示应用程序所需的配置位置进行了一些更改。对于设计用于与 v1.2.0 配合使用的 OTA 应用程序，要迁移到该库的 v3.0.0，您必须更新 PAL 移植功能签名并包含本迁移指南中所述的其他配置文件。

OTA PAL 的更改

- OTA PAL 移植目录名称已从 ota 更新为 ota_pal_for_aws。此文件夹必须包含 2 个文件：ota_pal.c 和 ota_pal.h。已从 OTA 库中删除 PAL 头文件 libraries/freertos_plus/aws/ota/src/aws_iot_ota_pal.h，并且必须在移植内定义。
- 将返回代码 (OTA_Err_t) 转换成了枚举 OTAMainStatus_t。有关转换后的返回代码，请参阅 ota_platform_interface.h。还提供了辅助标记宏，用于组合 OtaPalMainStatus 和 OtaPalSubStatus 代码，并从 OtaPalStatus 中提取 OtaMainStatus 以及类似的宏。
- 登录 PAL
  - 删除了 DEFINE_OTA_METHOD_NAME 宏。
  - 早期版本：OTA_LOG_L1("[%s] Receive file created.\r\n", OTA_METHOD_NAME);.
  - 更新版本：LogInfo(("Receive file created.")); 为相应的日志使用 LogDebug、LogWarn 和 LogError。
- 变量 cOTA_JSON_FileSignatureKey 已更改为 OTA_JsonFileSignatureKey。

函数

函数签名在 ota_pal.h 中定义并以前缀 otaPal 开头，而不是 prvPAL。

> **Note**
>
> 从技术上而言，PAL 的具体名称是开放式的，但为了满足资格认证测试要求，该名称应符合下面指定的规则。

- 版本 1：OTA_Err_t prvPAL_CreateFileForRx(OTA_FileContext_t * const *C*);

版本 3：OtaPalStatus_t otaPal_CreateFileForRx(OtaFileContext_t * const *pFileContext*);

备注：当数据块进入时，为其创建一个新的接收文件。

- 版本 1：int16_t prvPAL_WriteBlock(OTA_FileContext_t * const C, uint32_t ulOffset, uint8_t * const pcData, uint32_t ulBlockSize);

  版本 3：int16_t otaPal_WriteBlock(OtaFileContext_t * const pFileContext, uint32_t ulOffset, uint8_t * const pData, uint32_t ulBlockSize);

  备注：按照指定偏移量将数据块写入指定文件。

- 版本 1：OTA_Err_t prvPAL_ActivateNewImage(void);

  版本 3：OtaPalStatus_t otaPal_ActivateNewImage(OtaFileContext_t * const *pFileContext*);

  备注：激活通过 OTA 接收的最新 MCU 映像。

- 版本 1：OTA_Err_t prvPAL_ResetDevice(void);

  版本 3：OtaPalStatus_t otaPal_ResetDevice(OtaFileContext_t * const *pFileContext*);

  备注：重置设备。

- 版本 1：OTA_Err_t prvPAL_CloseFile(OTA_FileContext_t * const *C*);

  版本 3：OtaPalStatus_t otaPal_CloseFile(OtaFileContext_t * const *pFileContext*);

  备注：在指定 OTA 环境中验证并关闭底层接收文件。

- 版本 1：OTA_Err_t prvPAL_Abort(OTA_FileContext_t * const *C*);

  版本 3：OtaPalStatus_t otaPal_Abort(OtaFileContext_t * const *pFileContext*);

  备注：停止 OTA 传输。

- 版本 1：OTA_Err_t prvPAL_SetPlatformImageState(OTA_ImageState_t *eState*);

版本 3：OtaPalStatus_t otaPal_SetPlatformImageState(OtaFileContext_t * const pFileContext, OtaImageState_t eState);

备注：尝试设置 OTA 更新映像的状态。

- 版本 1：OTA_PAL_ImageState_t prvPAL_GetPlatformImageState(void);

  版本 3：OtaPalImageState_t otaPal_GetPlatformImageState(OtaFileContext_t * const *pFileContext*);

  备注：获取 OTA 更新映像的状态。

数据类型

- 版本 1：OTA_PAL_ImageState_t

  aws_iot_ota_agent.h 文件：

  版本 3：OtaPalImageState_t

  ota_private.h 文件：

  备注：平台实现设置的映像状态。
- 版本 1：OTA_Err_t

  aws_iot_ota_agent.h 文件：

  版本 3：OtaErr_t OtaPalStatus_t (combination of OtaPalMainStatus_t and OtaPalSubStatus_t)

  文件：ota.h、ota_platform_interface.h

  备注：v1：这些是定义 32 无符号整数的宏。v3：代表错误类型且与错误代码关联的专用枚举。
- 版本 1：OTA_FileContext_t

  aws_iot_ota_agent.h 文件：

  版本 3：OtaFileContext_t

  ota_private.h 文件：

备注：v1：包含枚举和数据缓冲区。v3：包含其他数据长度变量。

- 版本 1：OTA_ImageState_t

  aws_iot_ota_agent.h 文件：

  版本 3：OtaImageState_t

  ota_private.h 文件：

  备注：OTA 映像状态

配置更改

文件 aws_ota_agent_config.h 已重命名为 ota_config.h，这更改了从 _AWS_OTA_AGENT_CONFIG_H_ 到 OTA_CONFIG_H_ 的 include 保护。

- 文件 aws_ota_codesigner_certificate.h 已删除。
- 包括用于输出调试消息的新日志记录堆栈：

```
/************************************************/
/******* 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 ****************/
```

- 添加了常量配置：

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

新文件：ota_demo_config.h包含 OTA 演示所需的配置，例如，代码签名证书和应用程序版本。

- 在 demos/include/aws_ota_codesigner_certificate.h 中定义的 signingcredentialSIGNING_CERTIFICATE_PEM 已作为 otapalconfigCODE_SIGNING_CERTIFICATE 移至 ota_demo_config.h 中，并且可通过以下方式从 PAL 文件进行访问：

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

  文件 aws_ota_codesigner_certificate.h 已删除。
- 宏 APP_VERSION_BUILD、APP_VERSION_MINOR、APP_VERSION_MAJOR 已添加到 ota_demo_config.h 中。已删除包含版本信息的旧文件，例如 tests/include/aws_application_version.h、libraries/c_sdk/standard/common/include/iot_appversion32.h、demos/demo_runner/aws_demo_version.c。

OTA PAL 测试的更改

- 移除了“Full_OTA_AGENT”测试组以及所有相关文件。该测试组以前是资格认证所必需的。这些测试适用于 OTA 库，而不是特定于 OTA PAL 移植。现在，OTA 库具有托管在 OTA 存储库中的完整测试覆盖范围，因此不再需要此测试组。
- 删除了“Full_OTA_CBOR”和“Quarantine_OTA_CBOR”测试组以及所有相关文件。这些测试不是资格认证测试的一部分。这些测试所涵盖的功能现在正在 OTA 存储库中进行测试。
- 将测试文件从库目录移到了 tests/integration_tests/ota_pal 目录中。
- 更新了 OTA PAL 资格认证测试以使用 OTA 库 API 的 v3.0.0。
- 更新了 OTA PAL 测试访问测试代码签名证书的方式。以前，代码签名凭证有一个专用的头文件。对于新版本的库来说，情况不再是这样。测试代码预期在 ota_pal.c 中定义此变量。将该值分配给在平台特定的 OTA 配置文件中定义的宏。

核对清单

使用此核对清单确保遵循迁移要求的步骤：

- 将 ota pal 移植文件夹的名称从 ota 更新为 ota_pal_for_aws。
- 添加包含上述函数的文件 ota_pal.h。有关 ota_pal.h 示例文件，请参阅 GitHub。
- 添加配置文件：
  - 将文件名从 aws_ota_agent_config.h 更改为（或创建）ota_config.h。
    - 添加：

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

    - Include：

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

  - 将上述文件复制到 aws_test config 文件夹，并将 ota_demo_config.h 所有 Include 文件替换为 aws_test_ota_config.h。
  - 添加 ota_demo_config.h 文件。
  - 添加 aws_test_ota_config.h 文件。
- 对 ota_pal.c 进行以下更改：
  - 使用最新的 OTA 库文件名更新 Include 文件。
  - 删除 DEFINE_OTA_METHOD_NAME 宏。
  - 更新 OTA PAL 函数的签名。
  - 将文件上下文变量的名称从 C 更新为 pFileContext。
  - 更新 OTA_FileContext_t 结构和所有相关变量。
  - 将 cOTA_JSON_FileSignatureKey 更新为 OTA_JsonFileSignatureKey。
  - 更新 OTA_PAL_ImageState_t 和 Ota_ImageState_t类型。
  - 更新错误类型和值。
  - 更新打印宏以使用日志记录堆栈。
  - 将 signingcredentialSIGNING_CERTIFICATE_PEM 更新为 otapalconfigCODE_SIGNING_CERTIFICATE。
  - 更新 otaPal_CheckFileSignature 和 otaPal_ReadAndAssumeCertificate 函数注释。
- 更新 CMakeLists.txt 文件。

- 更新 IDE 项目。

文档历史记录

下表介绍《FreeRTOS 移植指南》和《FreeRTOS 资格指南》的文档历史记录。

日期	文档版本	更改历史记录	FreeRTOS 版本
2022 年 5 月	Amazon FreeRTOS 移植指南 FreeRTOS 资格认证指南	• 更新了现有测试，添加了新测试，并删除了基于 FreeRTOS 长期支持 (LTS) 库的冗余测试。有关更多信息，请参阅 GitHub 上的 FreeRTOS 库集成测试 202205.00。 • 已更新 FreeRTOS 移植流程图。 • 添加了一个新的 移植网络传输接口。 • 移植 AWS IoT over-the-air (OTA) 更新库 现在是资格认证所必需的。 • 删除了 Wi-Fi 和 TLS 抽象移植指南，因为不再需要这些文档。 • 有关 FreeRTOS 资格认证的更多更新，请参阅最新更改。	202012.04-LTS 202112.00
2021 年 7 月	202107.00（移植指南）	• 版本 202107.00	202107.00

日期	文档版本	更改历史记录	FreeRTOS 版本
	202107.00（资格认证指南）	• 更改了移植 AWS IoT over-the-air (OTA) 更新库 • 添加了为 OTA 应用程序从版本 1 迁移到版本 3 • 添加了为 OTA PAL 移植从版本 1 迁移到版本 3	
2020 年 12 月	202012.00（移植指南） 202012.00（资格认证指南）	• 版本 202012.00 • 添加了配置 coreHTTP 库 • 添加了移植蜂窝接口库	202012.00
2020 年 11 月	202011.00（移植指南） 202011.00（资格认证指南）	• 版本 202011.00 • 添加了配置 coreMQTT 库	202011.00
2020 年 7 月	202007.00（移植指南） 202007.00（资格认证指南）	• 版本 202007.00	202007.00
2020 年 2 月 18 日	202002.00（移植指南） 202002.00（资格认证指南）	• 版本 202002.00 • Amazon FreeRTOS 现在已更名为 FreeRTOS	202002.00

日期	文档版本	更改历史记录	FreeRTOS 版本
2019 年 12 月 17 日	201912.00（移植指南） 201912.00（资格认证指南）	• 版本 201912.00 • 添加了通用 I/O 库的移植	201912.00
2019 年 10 月 29 日	201910.00（移植指南） 201910.00（资格认证指南）	• 版本 201910.00 • 更新了随机数生成器移植信息。	201910.00
2019 年 8 月 26 日	201908.00（移植指南） 201908.00（资格认证指南）	• 版本 201908.00 • 添加了配置 HTTPS 客户端库以进行测试 更新了 移植 corePKCS11 库	201908.00
2019 年 6 月 17 日	201906.00（移植指南） 201906.00（资格认证指南）	• 发布版本 201906.00 • 更新了目录结构	201906.00 主版本
2019 年 5 月 21 日	1.4.8（移植指南） 1.4.8（资格认证指南）	• 移植文档移到了《FreeRTOS 移植指南》中 • 资格认证文档移到了《FreeRTOS 资格认证指南》中	1.4.8

日期	文档版本	更改历史记录	FreeRTOS 版本
2019 年 2 月 25 日	1.1.6	• 从“入门指南模板”附录中删除了下载和配置说明（第 84 页）	1.4.5 1.4.6 1.4.7
2018 年 12 月 27 日	1.1.5	• 使用 CMake 要求更新了“资格认证清单”附录（第 70 页）	1.4.5 1.4.6
2018 年 12 月 12 日	1.1.4	• 在 TCP/IP 移植附录中添加了 lwIP 移植说明（第 31 页）	1.4.5
2018 年 11 月 26 日	1.1.3	• 增加了低功耗蓝牙移植附录（第 52 页） • 在整个文档中添加了适用于 FreeRTOS 的 AWS IoT Device Tester 测试信息 • 在“列在 FreeRTOS 控制台上的信息”附录中添加了 CMake 链接（第 85 页）	1.4.4

日期	文档版本	更改历史记录	FreeRTOS 版本
2018 年 11 月 7 日	1.1.2	• 更新了 PKCS # 11 移植附录中的 PKCS # 11 PAL 接口移植说明（第 38 页） • 更新了 `CertificateConfigurator.html` 的路径（第 76 页） • 更新了“入门指南模板”附录（第 80 页）	1.4.3

日期	文档版本	更改历史记录	FreeRTOS 版本
2018 年 10 月 8 日	1.1.1	• 在 aws_test_runner_config.h 测试配置表中新增了“AFQP 所需”列（第 16 页） • 更新了“创建测试项目”部分中的 Unity 模块目录路径（第 14 页） • 更新了“建议的移植顺序”图表（第 22 页） • 更新了 TLS 附录“测试设置”中的客户端证书和密钥变量名称（第 40 页） • 更改了 Secure Sockets 库移植附录“测试设置”（第 34 页）、TLS 移植附录“测试设置”（第 40 页）和“TLS 服务器设置”附录（第 57 页）中的文件路径	1.4.2
2018 年 8 月 27 日	1.1.0	• 添加了 OTA 更新移植附录（第 47 页） • 添加了引导加载程序移植附录（第 51 页）	1.4.0 1.4.1

日期	文档版本	更改历史记录	FreeRTOS 版本
2018 年 8 月 9 日	1.0.1	• 更新了“建议的移植顺序”图表（第 22 页） • 更新了 PKCS # 11 移植附录（第 36 页） • 更改了 TLS 移植附录“测试设置”（第 40 页）和“TLS 服务器设置”附录第 9 步（第 51 页）中的文件路径 • 修补了 MQTT 移植附录“先决条件”中的超链接（第 45 页） • 为“创建 BYOC 的说明”附录中的示例添加了 AWS CLI 配置说明（第 57 页）	1.3.1 1.3.2
2018 年 7 月 31 日	1.0.0	《FreeRTOS 资格认证计划指南》的初始版本	1.3.0

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。