USER'S GUIDE

Apollo4 Family OEM Provisioning, Update, and Tools

Ultra-Low Power Apollo SoC Family

A-SOCAP4-UGGA01EN v1.6

Legal Information and Disclaimers

Revision History

Revision	Date	Description
1.0	October 10, 2020	Initial Release.
1.1	October 13, 2020	Added Debug Certificate chain picture.
1.2	October 22, 2020	Added OEM Asset Generation tool information.
1.3	December 21, 2020	Added new note in section 1 Introduction. Updated Content Certificate generation details in section 3.5 OEM Content Certificate Gen.
1.4	November 19, 2021	▪ Updated document template ▪ Added new content in section 1.1 System Requirements ▪ Added new section 2 Keys ▪ Updated section 3 OEM Provisioning ▪ Added new section 4 Customer Infospace Provisioning (INFO0) ▪ Updated section 5 OEM Image Certificate Generation ▪ Added new section 7 Image Generation for Apollo4 SBL ▪ Added new section 8 Downloading Images and Initiating Updates Through SBL ▪ Added new section 9 UART Wired Update ▪ Added new section 10Wired Download Procedure
1.5	October 19, 2022	▪ Updated document template
1.6	July 5, 2023	▪ Updated document title and section 1 Introduction

Reference Documents

Document ID	Description
A-SOCAP4-WPNA01EN	Apollo4 and Apollo4 Blue SoC Security Features White Paper
A-SOCAP4-UGGA02EN	Apollo4 Family Secure Update User’s Guide
	Apollo4 and Apollo4 Blue Secure Bootloader Scripts User’s Guide
	Apollo4 and Apollo4 Blue Getting Started Guide

Table of Contents

1. Introduction 9

1.1 System Requirements 9

1.2 Terminology 9

2. Keys 10

2.1 Key Gen Utility 11
- 2.1.1 Input Parameters 11
- 2.1.2 Key Gen Command 11

3. OEM Provisioning 12

3.1 HBK Gen Utility 13
- 3.1.1 Input Parameters 13
- 3.1.2 HBK Gen Command 13

3.2 OEM Key Request Utility 13
- 3.2.1 Input Parameters 13
- 3.2.2 OEM Key Request Command 14

3.3 OEM Asset Packaging Utility 14
- 3.3.1 Input Parameters 14
- 3.3.2 Command(s) For Asset(s) Generation 14

3.4 OEM Asset Gen Util 15
- 3.4.1 Input Parameters 15
- 3.4.2 Tool Execution 15

3.5 OEM Provisioning Data Gen Util 16
- 3.5.1 Input Parameters 16
- 3.5.2 Tool Execution 16

3.6 OEM Provisioning Tool (OPT) 16

4. Customer Infospace Provisioning (INFO0) 18

4.1 Info0 Generation 18

4.2 Info0 Programming 19

5. OEM Image Certificate Generation 20

5.1 Prerequisite 20

5.2 Note on the Tool Output Files 20

5.3 OEM Root Certificate Gen 21

5.3.1 Input Parameters 21
5.3.2 OEM Root Cert Gen Command 21
5.4 OEM Key Certificate Gen 21
5.4.1 Input Parameters 21
5.4.2 OEM Key Certificate Gen Command 22
5.5 OEM Content Certificate Gen 22
5.5.1 Input Parameters 22
5.5.2 OEM Content Certificate Gen Command 23

6. OEM Debug Certificate Generation 24

6.1 OEM Debug-Key Certificate Gen 25
6.1.1 Input Parameters 25
6.1.2 OEM Debug-Key Gen Command 25
6.2 OEM Debug Enabler Certificate Gen 25
6.2.1 Input Parameters 26
6.2.2 OEM Debug Enabler Certificate Gen Command 26
6.3 OEM Debug Developer Certificate Gen 26
6.3.1 Input Parameters 26
6.3.2 OEM Debug Developer Certificate Gen Command 26

7. Image Generation for Apollo4 SBL 28

7.1 Overview of Image Types 28
7.1.1 Firmware 28
7.1.2 Firmware OTA 29
7.1.3 Wired Download 29
7.1.4 Wired OTA 30
7.1.5 Summary 31
7.2 Image Generation Scripts 32
7.2.1 Basic Script Usage 32
7.2.2 Example Configuration File 33
7.2.3 Universal Security Options 34
7.3 Generating Specific Image Types 35
7.3.1 Firmware OTA Images 35
7.3.2 Wired Download Images 35
7.3.3 Info0 Update Images 36
7.3.4 OEM Certificate Chain Update 36
7.3.5 Key Revocation Images 36

8. Downloading Images and Initiating Updates 37

8.1 SWD Download Using JLINK 37
8.2 Wired Update 38

8.3 Over the Air Updates 38

9. UART Wired Update 40

9.1 Upgrading Multiple Images in One Step 41

9.2 Upgrading Large Binary (Using --wired-chunk-size feature) 41

10. Wired Download Procedure 43

10.1 Using Wired Download for OTA 43

List of Tables

Table 1-1 Terminology 9

List of Figures

Figure 3-1 General Flow for OEM Security Asset Generation 12
Figure 5-1 OEM Certificate Generation 20
Figure 6-1 OEM Debug Certificates 24
Figure 7-1 Firmware 28
Figure 7-2 Firmware OTA 29
Figure 7-3 Wired Download 30
Figure 7-4 Wired OTA 31
Figure 7-5 Wired OTA - Step 2 31
Figure 7-6 Relationship Summary 32

SECTION
1

Introduction

This document provides an overview of the OEM provisioning process, initial configuration, and run time updates for the Apollo4 Family SoC and the details of the tools required for the same. References to Apollo4 refers to the Apollo4 Family, unless explicitly stated otherwise.

NOTE: Unless specifically noted, all the tools mentioned below are included in the AmbiqSuite SDK release under directory **tools/apollo4b_scripts/** for Apollo4 and Apollo4 Plus, or **tools/apollo4l_scripts/** for Apollo4 Lite.

1.1 System Requirements

The provisioning tools are designed to run on a Linux host machine with the following requirements:

- Linux – Kernel Version 4.15.0 – 107-generic (Ubuntu 16.04.2).
- OpenSSL – 1.0.2g
- Python – 3.5.2
  - pyserial – Required for uart_wired_update.py
  - pycryptodome – Required for certain encrypted or signed image formats.

1.2 Terminology

This section defines some of the terminologies used in this document.

Table 1-1: Terminology

Abbreviation	Definition
ICV	Integrated Circuit Vendor
DM LCS	Device Manufacturer Life Cycle State
OEM	Original Equipment Manufacturer
RoT	Root of Trust
RMA	Returned Merchandise Authorization

SECTION

2 Keys

The Apollo4 Security infrastructure relies on a number of asymmetric and symmetric keys for variety of purpose, ranging from creating a Root of Trust, to using asymmetric keys for Authentication, and symmetric keys for various encryption needs.

- **Signing Keys**

  The Apollo4 uses Asymmetric PKA for establishing authenticity of images as well as the updates.

  A Secureboot enabled part requires a certificate chain for successful boot. The chain itself contains three certificates, each with a Public Key, with the root certificate binding to the OTP Root of Trust. Images/certificates are signed using the corresponding Private Keys. Key assets are maintained as Password encrypted .pem files.

- **Encryption Keys**

  The Apollo4 uses Symmetric AES-CTR encryption for code confidentiality. Symmetric key based AES-CMAC is also used to add authentication and encryption to provisioning information during manufacturing as well.

  OEMs can program their desired AES keys into OTP during the OEM provisioning process. Specifically, OEMs program the Apollo4 with a couple of hardware keys Kcp, a Kce, and a bank of additional AES keys (known as the keybank) in the OTP.

- **Key Generation**

  While the OEM may have their own key server to generate the key assets (e.g. HSM), AmbiqSuite SDK provides simple utilities that can be used to generate most of them as well.

2.1 Key Gen Utility

The Key Gen Utility is used to generate all the keys required for OEM secrets provisioning at the device manufacturing. The following python utility can be used to generate the keys.

```
./oem_tools_pkg/am_oem_key_gen_util/am_oem_key_gen_util.py
```

2.1.1 Input Parameters

The inputs to the Key Gen Utility are provided through the configuration file as a command-line parameter. The inputs configured in the configuration file is described in the example config file itself at the following location

```
./oem_tools_pkg/am_oem_key_gen_util/oemKeyGenConfig.cfg
```

2.1.2 Key Gen Command

The following are the Key Gen Command:

```
./oem_tools_pkg/am_oem_key_gen_util/$
python am_oem_key_gen_util.py ./oemKeyGenConfig.cfg
```

At successful execution, the Key Gen Utility should create the following two folders containing all the required symmetric and asymmetric keys for OTP provisioning. It also generates the asymmetric keys for OEM certificate chain and debug/RMA certificates.

```
./oem_tools_pkg/am_oem_key_gen_util/oemAesKeys
./oem_tools_pkg/am_oem_key_gen_util/oemRsaKeys
```

The keybank keys need to be generated separately.

SECTION

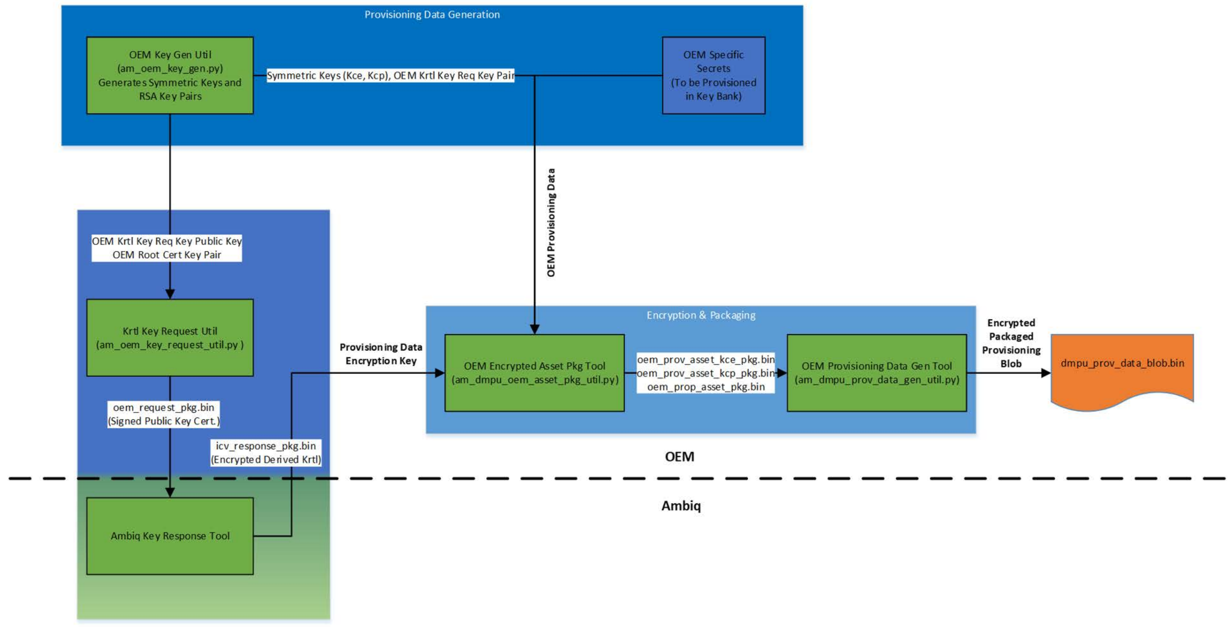
3

OEM Provisioning

OEM provisioning is the process of creating digital security assets in a form that is suitable for the customer's device production flow. Security assets include OEM symmetric keys, Root of Trust, and any security sensitive data provisioned to OTP at the time of device manufacturing.

Figure 3-1 shows the general flow for OEM security asset generation for the Apollo4 Family. The following sections outline the operation of the tools within each of blocks.

Figure 3-1: General Flow for OEM Security Asset Generation



Provisioning Data Generation
OEM Key Gen Util (am_oem_key_gen.py) Generates Symmetric Keys and RSA Key Pairs
Symmetric Keys (Kce, Kcp), OEM Krtl Key Req Key Pair
OEM Specific Secrets (To be Provisioned in Key Bank)
OEM Provisioning Data
OEM Krtl Key Req Key Public Key OEM Root Cert Key Pair
Krtl Key Request Util (am_oem_key_request_util.py)
oem_request_pkg.bin (Signed Public Key Cert.)
icv_response_pkg.bin (Encrypted Derived Krtl)
Provisioning Data Encryption Key
Encryption & Packaging
OEM Encrypted Asset Pkg Tool (am_dmpu_oem_asset_pkg_util.py)
oem_prov_asset_kce_pkg.bin oem_prov_asset_kcp_pkg.bin oem_prop_asset_pkg.bin
OEM Provisioning Data Gen Tool (am_dmpu_prov_data_gen_util.py)
Encrypted Packaged Provisioning Blob
dmpu_prov_data_blob.bin
OEM
Ambiq
Ambiq Key Response Tool

3.1 HBK Gen Utility

The HBK Gen Utility generates the Root Of Trust for the OEM (HBK 1).

```
./oem_tools_pkg/cert_utils/am_hbk_gen/am_hbk_gen_util.py
```

3.1.1 Input Parameters

The parameters to the HBK Gen Util are command-line parameters as shown below.

```
python am_hbk_gen_util.py -key <path to the OEM Root Public key> -
endian <B | L> -hash_format <SHA256 | SHA256_TRUNC >
```

3.1.2 HBK Gen Command

The command below will generate the truncated hash of the OEM Root Certificate Public key.

```
./oem_tools_pkg/cert_utils/am_hbk_gen/$
```

```
python am_hbk_gen_util.py -key ../../am_oem_key_gen_util/oemRSAKeys/
oemRootCertPublicKey.pem -endian L -hash_format SHA256_TRUNC
```

The output files will be generated at the following output folder:

```
./oem_tools_pkg/cert_utils/am_hbk_gen/hbk_gen_util_outPut
```

3.2 OEM Key Request Utility

The OEM Key Request Utility is used to generate the derived-Krtl-key-request-certificate to be processed at Ambiq’s secure lab. The output of this utility is sent to Ambiq which contains the key request public key. This is used by Ambiq’s response utility to encrypt the derived Krtl key used for encrypting the OEM assets.

```
./oem_tools_pkg/oem_asset_prov_utils/oem_key_request/
am_oem_key_request_util.py
```

3.2.1 Input Parameters

The inputs to the OEM Key Request Utility are provided through the configuration file as a command line parameter. The details of the config file parameters are described in the example config file itself.

```
./oem_tools_pkg/oem_asset_prov_utils/oem_key_request/am_config/
am_dmpu_oem_key_request.cfg
```

3.2.2 OEM Key Request Command

The following command will generate the key request binary data:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_key_request/$
python am_oem_key_request_util.py ./am_config/ am_dmpu_oem_key_re-
quest.cfg
```

The output file containing the key request data is generated as follows:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_key_request/oem_re-
quest_pkg.bin
```

The key request should be delivered to Ambiq which will process the request and return the **icv_response_pkg.bin** file.

3.3 OEM Asset Packaging Utility

The OEM Asset Packaging Utility is used to encrypt the OEM assets before sending it to the device manufacturing to provision the encrypted assets securely.

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/
am_dmpu_oem_asset_pkg_util.py
```

This utility mainly encrypts three plain text assets (Kcp, Kce, and OEM secrets) separately and generate three encrypted assets blobs. The plain text OEM-secrets and **icv_response_pkg.bin** received from Ambiq can be placed at the following location as described in the example config file:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/inputData/
```

3.3.1 Input Parameters

The three config files for each asset are as follows.

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config/
oem_asset_enc.cfg
```

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config/
am_asset_oem_ce.cfg
```

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config/
am_asset_oem_cp.cfg
```

3.3.2 Command(s) For Asset(s) Generation

The following command(s) are used to generate the encrypted OEM assets:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/$
```

```
python am_dmpu_oem_asset_pkg_util.py ./am_config/oem_asset_enc.cfg
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/$
python am_dmpu_oem_asset_pkg_util.py ./am_config/am_asset_oem_ce.cfg
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/$
python am_dmpu_oem_asset_pkg_util.py ./am_config/ am_as-
set_oem_cp.cfg
```

The output files containing the encrypted assets should be generated as follows.

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/
oem_asset_pkg.bin
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/
oem_prov_asset_kce_pkg.bin
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/
oem_prov_asset_kcp_pkg.bin
```

3.4 OEM Asset Gen Util

The OEM Asset Gen Utility generates a binary file used to initialize OEM OTP security settings. These setting are available to set in the file named **oem_asset_-gen.cfg**.

The tool creates a 2k binary data file that is an input to the OEM Provisioning Data Gen Utility discussed in the next section.

3.4.1 Input Parameters

OEM OTP security setting can be modified via the configuration file named **oem_asset_gen.cfg**:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/oem_asset_-
gen_util.py
```

The following config file is used to generate the OEM provision blob.

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config/
oem_asset_gen.cfg
```

3.4.2 Tool Execution

The following command will be used to generate the encrypted OEM assets blob:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/$
python oem_asset_gen_util.py ./am_config/oem_asset_gen_cfg
```

The output of the tool generates as follows:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/oem_as-
set_test_data.bin
```

3.5 OEM Provisioning Data Gen Util

The OEM Provisioning Data Gen Utility generates the final encrypted OEM provisioning blob which is decrypted by the OPT tool on the device before provisioning the data.

The tool mainly joins the three encrypted OEM assets generated by the OEM Asset Packaging Utility. It also adds default DCU, initial software version and more to the final blob which is provided through the config file.

The utility is available at the following path.

3.5.1 Input Parameters

The following is the input parameter:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/
am_dmpu_prov_data_gen_util.py
```

The following config file is used to generate the OEM provision blob:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config/
am_dmpu_data_gen.cfg
```

3.5.2 Tool Execution

The following command will be used to generate the encrypted OEM assets blob:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/$
python am_dmpu_prov_data_gen_util.py ./am_config/am_dmpu_data_-
gen_cfg
```

The output of the tool generates as follows:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/dmpu_prov_-
data_blob.bin
```

3.6 OEM Provisioning Tool (OPT)

The OPT is a Ambiq signed tool which is downloaded and executed on the chip during device production in the OEM manufacturing facility.

The tool is downloaded in SRAM at address 0x10030000, and the OEM encrypted assets generated as per *Section 3.5.2 Tool Execution on page 16*, at the SRAM address 0x10037000. After downloading these 2 blobs, the device needs to be reset. At the following boot, the OEM assets get provisioned if both the blobs are authenticated successfully. After a successful OEM provisioning, the device will reset, and is transitioned to secure LCS.

Note: If the device provisioning fails for some reason, it will remain in DM LCS.

The OPT tool is available at the following location:
`./oem_tools_pkg/oem_prov_tool/opt_image_pkg.bin`

SECTION

4

Customer Infospace Provisioning (INFO0)

Customer infospace, also known as INFO0, is a 2KB area of MRAM which controls a variety of customer-specific features of the MCU. The following sections provide information on how to provision INFO0 for a particular application.

The scripts for the following examples are also located in **/tools/apoll4b_scripts** directory in the AmbiqSuite SDK.

4.1 Info0 Generation

The script `create_info0.py` can be used to create a binary file to be populated as INFO0. It allows the user to define a number of other INFO0 parameters based on command line.

```
$ python3 create_info0.py --help
usage: create_info0.py [-h] [--valid {0,1,2}] [--version VERSION]
                       [--main MAINPTR] [--cchain CERTCHAINPTR]
                       [--trim CUSTTRIM] [--wTO WIREDTIMEOUT] [--u0 U0]
                       [--u1 U1] [--u2 U2] [--u3 U3] [--u4 U4] [--u5 U5]
                       [--sdcert SDCERT] [--rma RMAOVERRIDE] [--sresv SRESV]
                       [--loglevel {0,1,2,3,4,5}]
                       output
Generate Apollo4b Info0 Blob
positional arguments:
  output                Output filename (without the extension)
optional arguments:
  -h, --help            show this help message and exit
  --valid {0,1,2}       INFO0 Valid 0 = Uninitialized, 1 = Valid, 2 = Invalid
                        (Default = 1)?
  --version VERSION     version (Default = 0)?
  --main MAINPTR        Main Firmware location (Default = 0x18000)?
  --cchain CERTCHAINPTR
                        Certificate Chain location (Default = 0xFFFFFFFF)?
  --trim CUSTTRIM       customer trim ?
  --wTO WIREDTIMEOUT    Wired interface timeout in millisec (default = 20000)
  --u0 U0               UART Config 0 (default = 0xFFFFFFFF)
  --u1 U1               UART Config 1 (default = 0xFFFFFFFF)
  --u2 U2               UART Config 2 (default = 0xFFFFFFFF)
  --u3 U3               UART Config 3 (default = 0xFFFFFFFF)
  --u4 U4               UART Config 4 (default = 0xFFFFFFFF)
  --u5 U5               UART Config 5 (default = 0xFFFFFFFF)
  --sdcert SDCERT       Secure Debug Cert Address (default = 0x1ff400)
 --rma RMAOVERRIDE     RMA Override Config 2 = Enabled, 5 = Disabled (default
= 0x2)
  --sresv SRESV         SRAM Reservation (Default 0x0)
  --loglevel {0,1,2,3,4,5}
                        Set Log Level (0: None), (1: Error), (2: INFO), (4:
                        Verbose), (5: Debug) [Default = Info]
```

Example Usage:

Create INFO0 image with Wired UART set on pins 53 (Tx) and 55 (Rx) with baud 115200 (0x1C200), and the timeout of 5 Sec. Main image (for nonsecure boot) is expected at 0x18000, and the SD Cert location set to 0x1ff400. If configured for secureboot, the OEM cert chain is located at 0xC0000.

```
python3 ./create_info0.py --valid 1 --u0 0x1C200c0 --u1 0xFFFF3537 --u2 0x4
--u3 0x4 --u4 0x0 --u5 0x0 --main 0x18000  --version 1 --wTO 5000 --sdcert
0x1ff400 --cchain 0xc0000 info0
```

This will generate info0.bin, which can then be programmed to the device.

4.2 Info0 Programming

The generated info0 can be directly programmed using debugger through Boot-ROM provided helper functions, or a generic update flow provided by SBL can be used to update/program, Info0.

AmbiqSuite SDK provides a sample script jlink-prog-info0.txt for the direct programming of Info0 using debugger, using the first method.

This script can be processed by the JLINK command line utility with an invocation following the following format (for Windows):

```
JLink.exe -CommanderScript jlink-prog-info0.txt
```

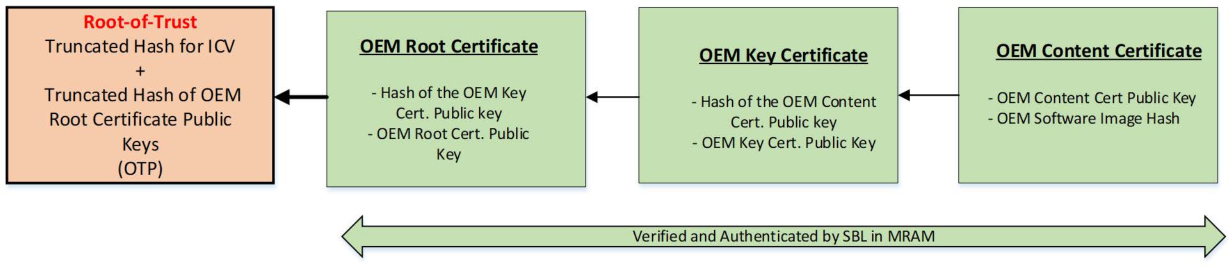
Alternatively, Info0 can be programmed using the SBL assisted update flow, by generating an Info0 update image (see *Section 7.3.3 Info0 Update Images on page 36*) and then using one of the supported methods to download the update image and initiate an update (see *Section 8 Downloading Images and Initiating Updates on page 37*).

SECTION 5

OEM Image Certificate Generation

Provisioning of the Apollo4 Family in the DM LCS depends upon a “chain” of public key certificates as shown in Figure 5-1. This method provides flexibility and additional security in case that the content changes or a certificate in the chain is compromised.

Figure 5-1: OEM Certificate Generation


Root-of-Trust
Truncated Hash for ICV
+
Truncated Hash of OEM Root Certificate Public Keys
(OTP)
OEM Root Certificate
- Hash of the OEM Key Cert. Public key
- OEM Root Cert. Public Key
OEM Key Certificate
- Hash of the OEM Content Cert. Public key
- OEM Key Cert. Public Key
OEM Content Certificate
- OEM Content Cert Public Key
- OEM Software Image Hash
Verified and Authenticated by SBL in MRAM

5.1 Prerequisite

The OEM certificate chain generation process assumes that OEM RSA keys have already been generated using the KeyGen Utility during the provisioning data generation process. The OEM Certificate chain should use these same OEM RSA keys.

While processing the Certificate Chain authentication on the device, it is also assumed that the device is already provisioned with the Root of Trust (HBK1 in the OTP), as the OEM Root Certificate needs to be authenticated using the RoT.

5.2 Note on the Tool Output Files

Many of the steps below will generate both ***.txt** files that are appropriate to cut and paste into source code files for testing. However, the tools also generate ***.bin** files that provide the assets for later steps to produce a provisioning blob.

5.3 OEM Root Certificate Gen

The OEM Root Certificate is used to validate the Public key provided by the OEM. It also authenticates the Public key embedded in the OEM key certificate, which is next in the chain.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util.py
```

5.3.1 Input Parameters

The inputs to the OEM Root Certificate Gen is provided through the configuration file as a command-line parameter. The details of the configuration file parameters are described in the following with example configuration values.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_config/am_oem_-
root_cert_hbk1.cfg
```

5.3.2 OEM Root Cert Gen Command

The following command generates the OEM Root Certificate:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/$
python am_cert_key_util.py ./am_config/am_oem_root_cert_hbk1.cfg
```

The output file containing the OEM Root Certificate is generated in binary and text formats:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output/
oem_root_cert_hbk1.bin
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output/
oem_root_cert_hbk1_Cert.txt
```

5.4 OEM Key Certificate Gen

The OEM Key Certificate Utility generates the OEM Key Certificate to validate the Public key in the Content Certificate which is next in the OEM Certificate Chain:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util.py
```

5.4.1 Input Parameters

The input to the OEM Key Certificate Gen is provided through the configuration file as a command-line parameter. The details of the configuration file parameters are described in the following with example configuration values.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_config/
am_oem_key_cert.cfg
```

5.4.2 OEM Key Certificate Gen Command

The following command generates the OEM Key Certificate:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/$
python am_cert_key_util.py ./am_config/am_oem_key_cert.cfg
```

The output file containing the OEM Key Certificate is generated in binary and text formats:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output/
oem_key_cert.bin
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output/
oem_key_cert_Cert.txt
```

5.5 OEM Content Certificate Gen

The OEM Content Certificate is used to authenticate OEM software images on the device. It contains a list of software images, along with the start addresses and their sizes.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_content_util.py
```

5.5.1 Input Parameters

The inputs to the OEM Content Certificate Gen Tool is provided through the configuration file as a command-line parameter. The details of the configuration file parameters are described in the following with example configuration values.

NOTE: Ambiq SBL mandates the software image(s) should be stored in MRAM un-encrypted and execute it from the stored location itself. Choose the corresponding configuration as described in the config file and set the **images_table.tbl** accordingly, as mentioned below. Please check with Ambiq before using other options.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_config/
am_oem_cnt_cert.cfg
```

The list of software images is listed in a text file that is used by the config file as mentioned above. The example list file is placed at the following location:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/inputData/images_table.tbl
```

The format of the image table are as follows:

- **ImageName** - Path to the image file.
- **RAMloadAdd** - Since the image is executed from the MRAM itself, as configured in config file, this must be set to image start address in MRAM.
- **flashStoreAdd** – Must be set to 0xFFFFFFFF.
- **Maxsize** - Must be set to a value equal to, or greater than the image size.
- **Enc-Scheme** - 0 - plain text, 1 encrypted.
  Note: The encrypted option is not supported and must not be selected.
- **WriteProtect** - When set to 0x1, Ambiq Bootloader will write protect the image (in 16K increments) as part of secure boot. Set it to 0, if no write protection is needed.
- **CopyProtect** - When set to 0x1, Ambiq Bootloader will copy protect the image (in 16K increments) as part of secure boot. Set it to 0, if no copy protection is needed. Note that, if enabled, the executable image must be built with no literals in the code area, as it will otherwise fail because of copy-protection.
- **ExtendedFormat** - Not Supported. Must be set to 0.

5.5.2 OEM Content Certificate Gen Command

The following command will be used to generate the content certificate:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/$
python am_cert_content_util.py ./am_config/am_oem_cnt_cert.cfg
```

The output file containing the Ambiq assets is generated as follows:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/
am_cert_content_util_output/content_cert.bin
./oem_tools_pkg/cert_utils/cert_gen_utils/
am_cert_content_util_output/content_cert_Cert.txt
```

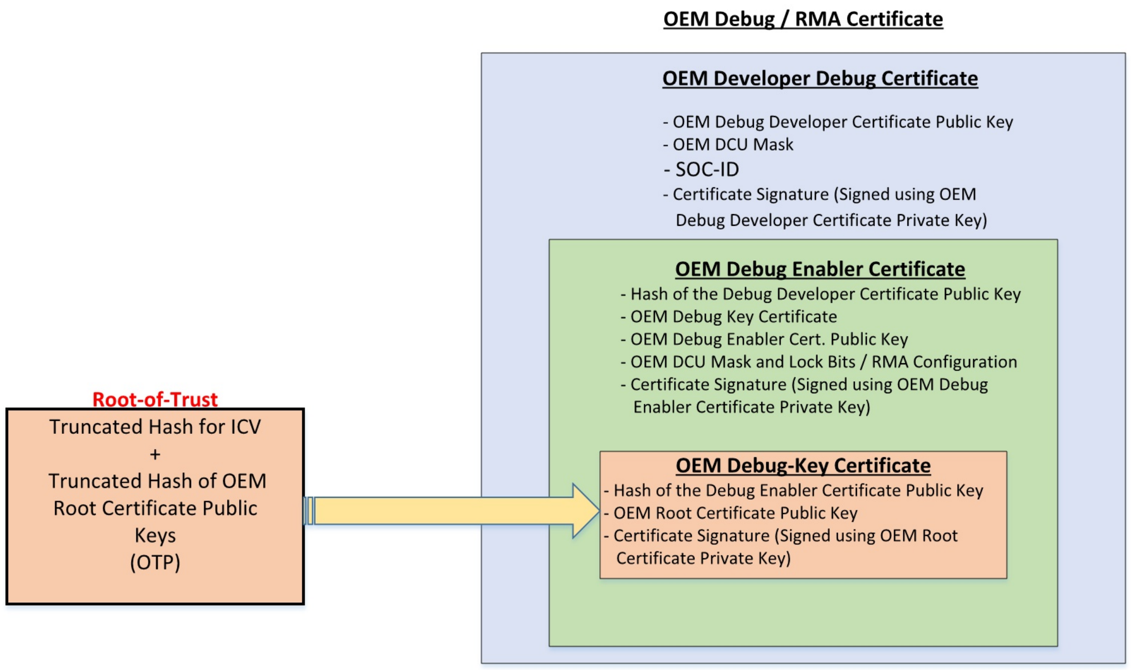
SECTION 6

OEM Debug Certificate Generation

The debug certificate(s) is used to enable the debugging or changing the device to RMA LCS for device analysis. The Debug certificates can be processed in DM and Secure LCS.

The debug certificate consists of three certificates debug-key certificate, Debug Enabler certificate, and Debug Developer certificate as shown in Figure 6-1. All these certificates need to be generated in the same sequence. The debug-key certificate is added to the debug enabler certificate using the Debug enabler certificate gen config file. Similarly, the debug enabler certificate is added to the debug developer certificate using the debug developer certificate gen config file. Finally, the debug developer certificate is downloaded on the device as a single entity to be processed by the SBR as three combined certificates.

Figure 6-1: OEM Debug Certificates


OEM Debug / RMA Certificate
OEM Developer Debug Certificate
- OEM Debug Developer Certificate Public Key
- OEM DCU Mask
- SOC-ID
- Certificate Signature (Signed using OEM Debug Developer Certificate Private Key)
OEM Debug Enabler Certificate
- Hash of the Debug Developer Certificate Public Key
- OEM Debug Key Certificate
- OEM Debug Enabler Cert. Public Key
- OEM DCU Mask and Lock Bits / RMA Configuration
- Certificate Signature (Signed using OEM Debug Enabler Certificate Private Key)
OEM Debug-Key Certificate
- Hash of the Debug Enabler Certificate Public Key
- OEM Root Certificate Public Key
- Certificate Signature (Signed using OEM Root Certificate Private Key)
Root-of-Trust
Truncated Hash for ICV + Truncated Hash of OEM Root Certificate Public Keys (OTP)

6.1 OEM Debug-Key Certificate Gen

The OEM Debug-Key Certificate is used to validate the Public key provided by the OEM. It also authenticates the Public key embedded in the OEM Enabler Certificate, which is next in the debug certificate chain.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util.py
```

6.1.1 Input Parameters

The inputs to the OEM Debug-Key Certificate Gen is provided through the configuration file as a command-line parameter. The details of the configuration file parameters are described in the following with example configuration values.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_config /am_oem_db-
g_key_cert.cfg
```

6.1.2 OEM Debug-Key Gen Command

The following command generates the OEM Debug-Key Certificate:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/$
python am_cert_key_util.py ./am_config/am_oem_dbg_key_cert.cfg
```

The output file containing the ICV Debug-Key Certificate is generated in binary and text formats as follows.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output/
debug_oem_key_cert.bin
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output/
debug_oem_key_cert_Cert.txt
```

6.2 OEM Debug Enabler Certificate Gen

The OEM Debug Enabler Certificate utility is used to generate the debug enabler certificate which contains the OEM DCU debug masks, and lock masks, or RMA information (in case of RMA certificate for OEM). It also authenticates the Public key embedded in the OEM debug developer certificate, which is next in the debug certificate chain.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_dbg_en-
abler_util.py
```

6.2.1 Input Parameters

The inputs to the OEM Debug Enabler Certificate Gen is provided through the configuration file as a command-line parameter. The details of the configuration file parameters are described in the following with example configuration values.

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_config/
am_oem_dbg_enabler_cert.cfg
```

6.2.2 OEM Debug Enabler Certificate Gen Command

The following command generates the OEM Debug Enabler Certificate:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/$
python am_cert_dbg_enabler_util.py ./am_config/am_oem_dbg_en-
abler_cert.cfg
```

The output file containing the OEM Debug Enabler Certificate is generated in binary format:

```
./oem_tools_pkg /cert_utils/cert_gen_utils/ am_debug_cert_output/
oem_dbg_cert_enabler_pkg.bin
```

6.3 OEM Debug Developer Certificate Gen

The OEM Debug Developer Certificate is used to generate the final debug certificate which contains OEM debug-key Certificate, OEM Debug Enabler Certificate, and SOC-ID (Chip specific Identification).

```
./oem_tools_pkg /cert_utils/cert_gen_utils/am_cert_dbg_devel-
oper_util.py
```

6.3.1 Input Parameters

The inputs to the OEM Debug Developer Certificate Gen is provided through the configuration file as a command-line parameter. The inputs configured in the configuration file is described in the example configuration file itself at the following location:

```
./oem_tools_pkg /cert_utils/cert_gen_utils/am_config/
am_oem_dbg_developer_cert.cfg
```

6.3.2 OEM Debug Developer Certificate Gen Command

The following command generates the OEM Developer Certificate:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/$
```

```
python am_cert_dbg_developer_util.py ./am_config/
am_oem_dbg_developer_cert.cfg
```

The output file containing the OEM Debug Developer Certificate is generated in binary and text ('c' Header file) formats as follows:

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_debug_cert_output/
oem_dbg_cert_developer_pkg.bin
```

```
./oem_tools_pkg/cert_utils/cert_gen_utils/am_debug_cert_output/
oemDeveloperCert.h
```

SECTION

7

Image Generation for Apollo4 SBL

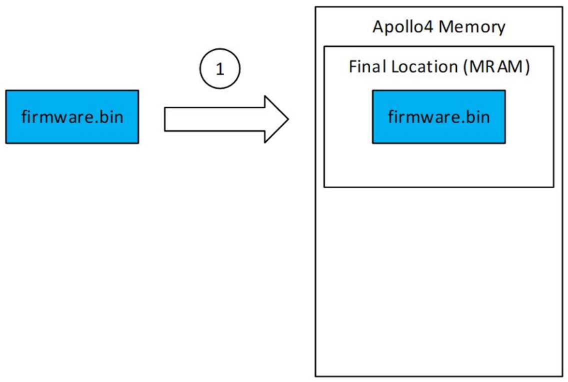
The Apollo4's boot ROM and SBL (secure boot loader) are capable of performing a wide variety of functions, each of which requires a binary image with a specific format, which in turn must be loaded using a specific protocol. This section provides an overview of the various types of binaries supported by the SBL, as well as some information about the scripts used to generate them.

7.1 Overview of Image Types

7.1.1 Firmware

Raw application binaries are the most basic image format for the Apollo4. This is simply a compiled application in the standard ARM format, and it can be loaded directly to MRAM using any SWD programming tool (including the JLINK device on the Apollo4 evaluation kits). All later firmware images will be derived from this basic image.

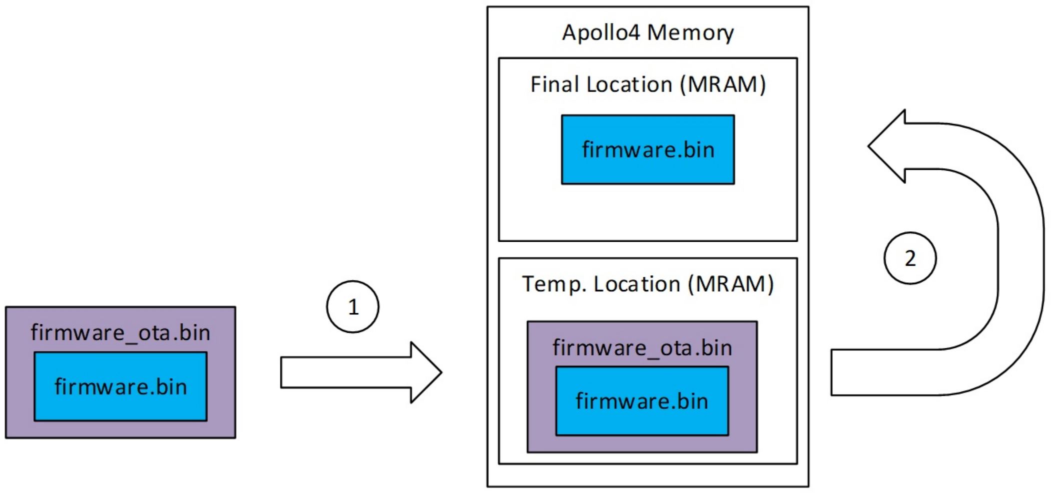
Figure 7-1: Firmware



7.1.2 Firmware OTA

Firmware OTA images are created by adding metadata to raw binaries using the create_cust_image_blob.py script (described in detail later in this document). OTA images are first loaded into a temporary location in MRAM using SWD programming tools (step 1) or through other application specific means for field upgrade (e.g., Firmware upgrade over BLE), and then the SBL will validate and load the inner application binary to its final destination (step 2). OTA images provide the user with the option to encrypt and/or sign application binaries. The most common use case for an OTA image is for a firmware upgrade. An existing user application can receive a binary (optionally encrypted or signed) from an external source, program that binary to a location, and then instruct the SBL to finish the firmware upgrade following a reset, potentially replacing the original user application.

Figure 7-2: Firmware OTA



To make this two-step process easier during development, AmbiqSuite includes a set of JLink scripts demonstrating how to load the OTA image to an appropriate temporary location over SWD and also how to trigger the SBL to perform the second step of the upgrade. See the **tools/apollo4b_scripts** directory in AmbiqSuite to find these scripts. The script also serves as a reference for the process an application would need to follow the over-the-air upgrades when using alternative means to download the OTA image.

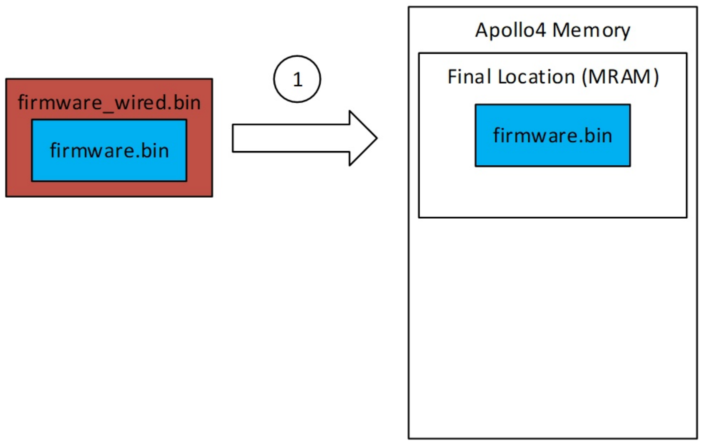
The same OTA process is reused for non-firmware upgrades like INFO0, trim patches, key revocations, etc., as well.

7.1.3 Wired Download

When using SBL provided Wired Download functionality, the raw images need to be formatted for SBL to understand and process.

Similar to OTA images, wired download images are also created by adding meta-data to raw binaries with the `create_cust_image_blob.py` script. Wired download metadata can be used to wrap any image type, and it will change how the image may be downloaded and stored/processed on the device. The wired download image format is understood both by the SBL and by the PC-side utility `uart_wired_update.py`. Using this utility and a wired download image, a user can effectively program the Apollo4 MRAM using a UART instead of an SWD connection. This is helpful in situations where an SWD connection is unavailable, including situations where an open SWD port might be a security risk. Depending on device configuration, the wired download could provide a secure (encrypted and authenticated) channel for device programming/updates. The wired download is fully managed by the SBL, so this procedure can be used on an otherwise unprogrammed the Apollo4 device. The SBL will read the wired download data directly over the UART interface, potentially decrypting it or authenticate using an RSA signature, and program the data to its final location in MRAM. No temporary MRAM is required for this process.

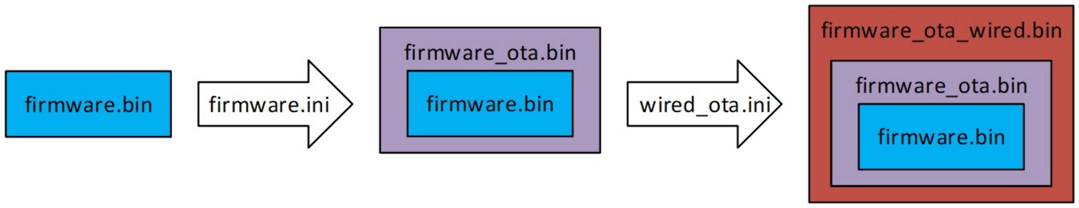
Figure 7-3: Wired Download



7.1.4 Wired OTA

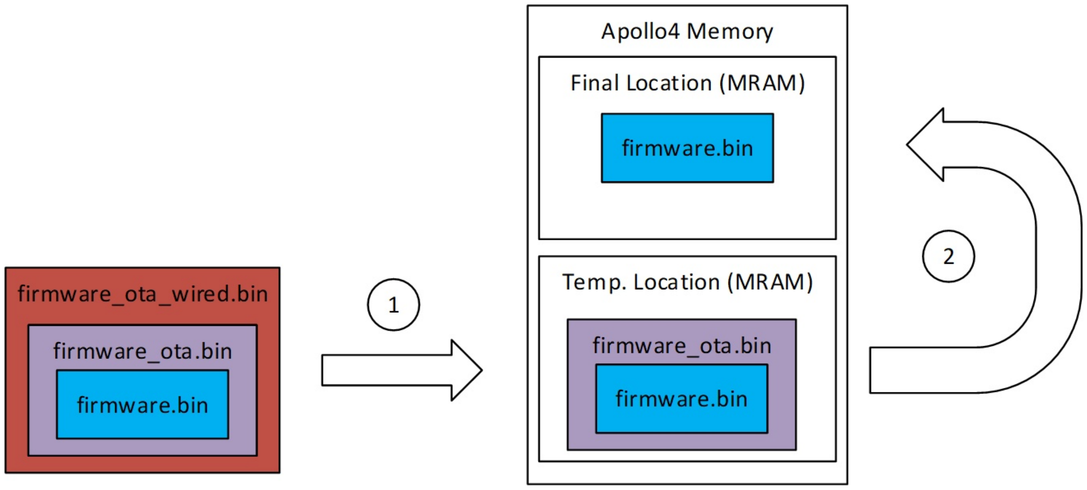
The OTA and Wired Download formats from the previous sections can be used simultaneously for a single image. In this case, the user will first process the raw binary to create an OTA image, and then process the OTA image to create a Wired OTA image. This format combines the advantages of the two formats to create an image that can be loaded entirely over UART, but which can also be loaded to a temporary location to avoid corrupting a known good image until the update is fully decrypted and authenticated.

Figure 7-4: Wired OTA



The update process with a Wired OTA image is similar to the standard OTA process, but image download step is performed over UART instead of SWD. In step one, the UART boot host will send the Wired OTA image to the SBL, which will program its contents (the OTA image) to a temporary location in MRAM. Afterwards, the Apollo4 will reboot and perform step 2, where it processes the OTA image and programs the device firmware to its final location.

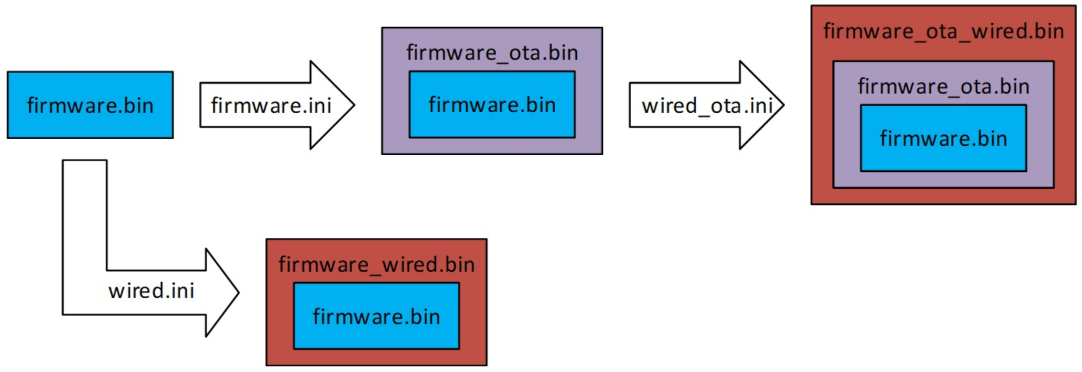
Figure 7-5: Wired OTA - Step 2



7.1.5 Summary

The diagram below further summarizes the relationship between raw binaries, OTA images, Wired Download images, and Wired OTA images. While the examples used in the previous sections all assumed we were working with an application binary, there are other image types that can be used in place of the OTA image type shown here.

Figure 7-6: Relationship Summary



The following sections discuss these additional image types and also explain in detail how the supporting scripts work.

7.2 Image Generation Scripts

The Apollo4's SBL (secure boot loader) is capable of performing a wide variety of functions, each of which requires a binary image with a specific format, which in turn must be loaded using a specific protocol. The `create_cust_image_blob.py` script described in this section can generate most of these image formats, including the following:

- Secure and non-secure Firmware OTA
- Wired Download
- Certificate Chain Update
- Key Revocation
- INFO0 Update

7.2.1 Basic Script Usage

Most image formats for the Apollo4 are handled by `create_cust_image_blob.py`. This script can be controlled either with command line parameters, by config file, or with a combination of the two, where the command-line arguments always take precedence. The full list of options can be found by calling the script using the --help command line option, but many options are not required for basic images. The following sections describe the supported image types along with the relevant options for each one.

For more examples of how to use `create_cust_image_blob.py`, see the **tools/apollo4b_scripts/examples** directory of AmbiqSuite.

7.2.2 Example Configuration File

Below is an example showing the configuration files used for `create_cust_image_blob.py` alongside the equivalent command-line commands.

Configuration file "firmware.ini":

```
#****************************************************************************
#
# Configuration file for create_cust_image_blob.py
#
# Run "create_cust_image_blob.py --help" for more information about the
options
# below.
#
# All numerical values below may be expressed in either decimal or hexadeci-
mal
# "0x" notation.
#
# To re-generate this file using all default values, run
# "create_cust_image_blob.py --create-config"
#
#****************************************************************************
[Settings]
chip = apollo4b
app_file = hello_world.bin
# Location where the image should be installed
load_address = 0x18000
enc_algo = 0x0
# specify enc_algo as 1 to do AES encryption
auth_algo = 0x0
# auth_key relevant only if auth_algo is 1
# auth_key indicates which PK is used for signature
auth_key = 0x2
kek_index = 0x80
image_type = firmware
certificate = None
output = hello_world_ota.bin
key_table = keys.ini
```

This configuration creates hello_world_ota.bin (a non-secure firmware OTA image) from hello_world.bin (a compiled Apollo4 binary), with no encryption, authentication, or content certificate. To execute this configuration, one would run the following command from the same directory as the file.

```
$ python3 ./create_cust_image_blob.py -c firmware.ini
```

Equivalently, this configuration can also be expressed on the command line as follows.

```
$ python3 ./create_cust_image_blob.py --chip apollo4b --bin hel-
lo_world.bin --load-address 0x18000 --enc-algo 0 --kek-index 0x80 --
auth_algo 0 --auth-key 0x2 --image-type firmware
```

7.2.3 Universal Security Options

The Apollo4 SBL supports AES encryption and RSA authentication for a wide variety of update types. Correspondingly, the `create_cust_image_blob.py` script includes features to encrypt and/or sign binaries in the correct format to be decrypted and validated by the SBL. Each of the security options below can be applied to any image type. Additional options for specific image types will be covered in later sections.

- **auth_algo:** The authentication method to use. (0 = none, 1 = RSA (RSA PSS 2072 after Hash SHA 256))
- **auth_key**: The key index of the authentication key. The index represents the actual asymmetric key used for signing. It corresponds to public key contained in one of the preinstalled certificates on the device (0 == root cert, 1 == key cert, 2 == content cert).
- **enc_algo**: The encryption algorithm to use for securing the transfer (0 = none, 1 = AES-128 CTR mode)
- **kek_index**: The index for the key encryption key to be used. The index represents the key-encryption-key used to decrypt the encrypted key bundled in the image. Index 0 and 1 represent hardware keys (Kcp or Kce respectively) programmed during provisioning. Key indices 0x80 onwards represent OTP keybank keys, with each index representing a 128b AES key.
- **key_table**: Path to a configuration file describing the encryption keys used with the Apollo4. (Can be set to "None" if no encryption or authentication is needed)

This **key_table** file contains pointers to various Private Key assets used to sign the image blobs, as well as the symmetric keys used for encryption.

The asymmetric key material should correspond to the certificates provisioned on the chips, and are referenced using password encrypted **.pem** files.

The symmetric key material should match with the OTP keys programmed in the chips, and is referenced using pointers to binary key and keybank files. Typically, these keys will be used as Key Encryption Keys (KEK) for an OTA image.

To use a particular key, you will need to supply the **keys.ini** file as shown above, as well as a KEK index for encryption and Auth Keys index for signing.

```
# Key file
[Root Key]
index = 0x0
format = pem
filename = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/oemRootCertKey-
Pair.pem
passfile = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/pwdOemRootCert-
Key_Rsa.txt

[Key Cert Key]
index = 0x1
format = pem
```

```
filename = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/oemKeyCertKey-
Pair.pem
passfile = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/pwdOemKeyCert-
Key_Rsa.txt

[Content Cert Key]
index = 0x2
format = pem
filename = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/oemContentCertKey-
Pair.pem
passfile = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/pwdOemContentCert-
Key_Rsa.txt


[Symmetric Keys]
kcp = ../../oem_tools_pkg/am_oem_key_gen_util/oemAESKeys/kcp.bin
kce = ../../oem_tools_pkg/am_oem_key_gen_util/oemAESKeys/kce.bin
kb0 = oem_tools_pkg/oem_asset_prov_utils/inputData/keyBank0.bin
kb1 = oem_tools_pkg/oem_asset_prov_utils/inputData/keyBank1.bin
kb2 = oem_tools_pkg/oem_asset_prov_utils/inputData/keyBank2.bin
kb3 = oem_tools_pkg/oem_asset_prov_utils/inputData/keyBank3.bin
```

7.3 Generating Specific Image Types

7.3.1 Firmware OTA Images

Firmware OTA images contain firmware and metadata to be used by the bootloader to update the firmware on the device. Firmware images can also be created with built-in encryption and authentication using the "secure-firmware" image type. The following options are relevant for firmware and secure-firmware images:

- **app_file**: File to be used as the application binary
- **certificate**: For secure images, the binary file corresponding to the content certificate to be installed alongside the application binary.
- **image_type**: set to "firmware" for non-secure images, or "secure-firmware" for secure images.
- **load_address**: Address where the application should be written in the MRAM.

7.3.2 Wired Download Images

Wired download images can contain any blob and instructions for programming to a specific location. The bootloader will read the image and program the blob to the correct location. Wired images can be created with or without built-in encryption using the "wired" image type. Once the wired image is created, the `uart_wired_update.py` script can be used to execute the wired download instructions and load the blob into the Apollo4 memory. Wired download images support the following options:

- **app_file**: Binary file to write to internal memory

- **image_type**: set to “wired”
- **load_address**: Address where the binary data should be written in the MRAM.
- **ota**: Set this bit to alert the SBL that this wired download contains a firmware OTA.
- **wired_chunk_size**: Largest number of bytes the Apollo4 should download in a single pass. See wired OTA description in *Section 9.1 Upgrading Multiple Images in One Step on page 41*.

7.3.3 Info0 Update Images

This image type can be used to update the INFO0 data in the Apollo4 device. INFO0 is used to contain certain non-volatile Apollo4 device settings. For more information, see the `create_info0.py` script documentation in *Section 4.1 Info0 Generation on page 18*. INFO0 could be updated as a whole, or partially.

- **app_file**: Binary file to write to info0 space
- **image_type**: set to “info0”
- **offset**: The word offset within info0 where this update binary should be written.

7.3.4 OEM Certificate Chain Update

OEM Chain Update images are used to make updates to the certificate chain used by the Apollo4 SBL. The root certificate, key certificate, and content certificate all gets changed using this process. This option will be primarily used to increase the certificate version to ensure images cannot be rolled back to older certificates.

- **certificate**: Filename for the content certificate to be installed.
- **root_cert**: Filename for the root certificate to be installed.
- **key_cert**: Filename for the key certificate to be installed.

7.3.5 Key Revocation Images

The Apollo4 “keybank” keys (as discussed in *Section 1 Keys on page 10*) are programmed into OTP and can be used for encryption both in the boot process and in the main application. The purpose of the “key revocation” image type is to give users a way to revoke access to specific keybank keys in case they have been compromised.

In addition to the universal options, the key revocation image type supports only the following option:

- **app_bin**: Filename for the key update binary to be used for key revocation.

The key update binary itself is a 64-bit bitmask (little endian), where each bit corresponds to a single 128-bit AES key from the keybank. Setting a bit revokes the key with the same number (for example, setting bit 0 revokes keybank AES key 0).

SECTION

8

Downloading Images and Initiating Updates

As described in the previous chapter, the Apollo4 SBL is capable of receiving and validating firmware updates through multiple methods. This chapter focuses on the available methods for transferring data from a boot host (often a PC) to the Apollo4 SBL. The currently supported data transfer methods are as follows:

- SWD Debugger (such as JLink)
- Wired Update
- Over the Air (with the help of a user application)

AmbiqSuite contains examples for each of these transfer methods, and any of them may be used for firmware updates in customer applications. The following sections describe the available examples in further detail.

8.1 SWD Download Using JLINK

The most straightforward firmware download method is over SWD with a hardware debugger (like the JLINK). Using the debugger, a user may write directly to MRAM or INFO0. Depending on the type of image downloaded, the SBL will then take over the image validation and boot process immediately following the next reset operation.

AmbiqSuite includes JLINK scripts that show how the JLINK can be used to program a specific memory location in the Apollo4, and initiate an update through SBL.

These scripts include:

- **General Update**
  - jlink-ota.txt: An example of a debugger-driven OTA download. This can be used with any update image except for SBL update
- **SBL Update**

Installation of a replacement Secure Bootloader (SBL) requires special processing. There are two slots (at addresses 0x00008000 & 0x00010000) which are reserved for Secure Bootloader images and their corresponding Certificate Chains. Only one image is in use at a time.

The SBL Image Update consists of two blobs

- SBL Metadata (**sbl_ota.bin**) which can be stored anywhere in MRAM and is discarded after the update
- Encrypted SBL Image (**encrypted_sbl0.bin**) which needs to be stored at the "Staging Area"

The SBL image update is initiated similar to other images, and consists of downloading the Encrypted SBL image to staging area, and then downloading the SBL metadata just like any other OTA blob and initiating the OTA.

AmbiqSuite SDK provides reference JLink scripts for SBL updates in **tools/apollo4b_scripts/jlink-prog-sbl[01].txt**:

- jlink-prog-sbl1.txt to be used if current SBL running from 0x8000
- jlink-prog-sbl0.txt to be used if current SBL running from 0x10000

Each of these scripts can be processed by the JLINK command line utility with an invocation following the following format (for Windows): `JLink.exe -CommanderScript jlink-ota.txt`

8.2 Wired Update

Wired updates are a way to send data to the SBL without using a debugger connection. The SBL contains a UART or SPI-based boot client that can receive firmware updates from a host application. The UART boot host program is called uart_wired_update.py, and is discussed further in the next chapter.

8.3 Over the Air Updates

The SBL can also be used to support over the air updates enabled by user firmware. In this use case, the user firmware will communicate with a boot host and download an OTA image (described in the image formats section). Once the download is completed, the user applica- tion can program the update blob information by programming the OTA infrastructure (see *Apollo4 Family Secure Update User's Guide*), and initiate a device reset. SBL takes over on next bootup to validate the downloaded image, and perform a firmware upgrade

The AmbiqSuite SDK provides an example of an OTA firmware update over BLE through the "AMOTA" app for the Apollo4. This example implements a specific transfer protocol with a counterpart host implemented as a Phone App (Ambiq_BLE App).

The **`ota_binary_converter.py`** script in tools/apollo4_amota/scripts can be used to generate an OTA blob compatible with AMOTA. Most of the optional parameters are no longer relevant for the Apollo4.

Example usage:

```
python3 ota_binary_converter.py --appbin main_ota.bin -o main_ ota_a-
mota
```

Thereafter, the normal procedure to upgrade the image using AMOTA and Ambiq_BLE App on the phone can be used to upgrade the firmware on the device.

SECTION 9 UART Wired Update

For UART based wired update to work, the device needs to be provisioned to allow UART wired update through OTP and InfoSpace settings (see *Apollo4 SoC Security User's Guide*). SBL will get into update mode in one of the two cases:

- Encountering Boot error (e.g., invalid main image)
- GPIO Override (configured through OTP)

The host needs to be connected to the device on the configured pins to match with the Info-Space UART configurations, and needs to initiate the communication within a short window configured (through InfoSpace).

Script **`uart_wired_update.py`** is designed to emulate the host side functions in a limited way when using the UART as wired interface. This can be useful during development to test the UART wired update features, and to program the Apollo4 device in select ways while the debugger is disabled. Usage information for **`uart_wired_update.py`** appears below:

```
$ python3 uart_wired_update.py --help

usage: uart_wired_update.py [-h] [-b BAUD] [--raw RAW] [-f BINFILE] [-o OTADESC] [-r
{0,1,2}] [-a {0,1,-1}] port

UART Wired Update Host for Apollo4b

positional arguments:

  port         Serial COMx Port

optional arguments:

  -h, --help   show this help message and exit

  -b BAUD      Baud Rate (default is 115200)

  --raw RAW    Binary file for raw message

  -f BINFILE   Binary file to program into the target device

  -o OTADESC   OTA Descriptor Page address (hex) - (Default is 0xFE000) - enter
0xFFFFFFFF to instruct SBL to skip OTA

 -r {0,1,2}   Should it send reset command after image download? (0 = no reset, 1 = POI,
2 = POR) (default is 1)
```

```
  -a {0,1,-1}  Should it send abort command? (0 = abort, 1 = abort and quit, -1 = no
abort) (default is -1)
```

Example usage: Downloading an application using the wired download protocol.

```
python3 uart_wired_update.py -b 115200 COM3 -o 0xFFFFFFFF -r 0 -f application_wired.bin
```

Example usage: Downloading an OTA image of an application binary using the wired protocol.

```
python3 uart_wired_update.py -b 115200 COM3 -r 0 -f application_wired_ota.bin.
```

9.1 Upgrading Multiple Images in One Step

SBL supports upgrading multiple images in a single upgrade cycle using multiple entries in OTA Descriptor.

UART Wired Update scripts can be used to achieve the same. The script is to be run multiple times, once for each image. The key here is that OTA Descriptor is to be set only in the first invocation, and reset is to be issued only for the last one.

Example below shows upgrading an isolated data segments and main image (all considered non-secure main images generated as in section 6.1.1) together using uart_wired_update.py:

First image (also programs the OTA Descriptor, and does not reset the device):

```
python3 uart_wired_update.py -b 115200 COM<X> -f img1_nonse-
cure_wire.bin -i 6 -r 0
```

Second image (does not program the OTA Descriptor or reset the device):

```
python3 uart_wired_update.py -b 115200 COM<X> -f img2_nonse-
cure_wire.bin -i 6 -r 0 –o 0xFFFFFFFF
```

Third image (does not program the OTA Descriptor but resets the device to initiate the upgrade):

```
python3 uart_wired_update.py -b 115200 COM<X> -f img3_nonse-
cure_wire.bin -i 6 -r 1 –o 0xFFFFFFFF
```

9.2 Upgrading Large Binary (Using --wired-chunk-size feature)

SBL uses DTCM for local reassembly and processing of wired update messages to ensure only validated images are written to MRAM. The SBL reserves a fixed amount of memory for its own operation, reducing the total amount of DTCM available for wired updates. User applications may also require some portions of DTCM to remain intact, further limiting the available space for updates. To allow for wired downloads greater than the size of the available DTCM, the wired download procedure can be made to download a large image in smaller chunks using the --wire-chunk-size option.

To split a wired download into sections, you can specify the --wire-chunk-size option in the wired download configuration file with the maximum number of bytes of scratch space available in the Apollo4 MCU. The **`create_cust_image_blob.py`** script will automatically split the OTA image into a series of Wired Download images wrapped into a single binary. You can then download this binary to the Apollo4 device using the **`uart_wired_update.py`** script as normal. The script will detect that the wired update image is actually composed of multiple pieces and perform the download accordingly.

SECTION 10

Wired Download Procedure

The standard use of wired download images is to allow a binary to be programmed to a device using a method other than direct programming using Serial Wire Debug (SWD). This can be useful when the SWD interface is either disabled or physically inaccessible. The basic procedure for downloading a binary to the Apollo4 memory using the wired download option is as follows:

1. Use `create_cust_image_blob.py` to wrap the binary (shown here as "application.bin") into a wired download image. Here, the input file "**wired.ini**" contains information about where the application binary should be downloaded.

   ```
   $ python3 create_cust_image_blob.py -c wired.ini --app_file application.bin
   --output application_wired.bin
   ```

2. Use the resulting **application_wired.bin** file with the `uart_wired_update.py` script to load the application into the Apollo4 memory. This will place a copy of application.bin directly into the Apollo4 memory at the address specified in **wired.ini**.

10.1 Using Wired Download for OTA

A common use of the "wired download" format is to program a firmware OTA image into a temporary location in the Apollo4 memory, and then have the SBL validate it and move it to its final location. Some reasons you might favor this approach over the previous approach include:

- You are using a secure device, and you need to install a matching content certificate alongside your application binary. (OTA files allow the inclusion of content certificates)
- You want the binary data to be encrypted during the download portion.
- You want the binary data to be authenticated by the SBL before being installed.

For most wired download OTA operations, you can use the following procedure:

1. Use `create_cust_image_blob.py` to wrap an application binary into an OTA blob. Here, **firmware.ini** contains the relevant settings for generating the OTA image, and command line options are used to set the input and output binary names.

   ```
   $ python3 create_cust_image_blob.py -c firmware.ini --app_file
   application.bin --output application_ota.bin
   ```

2. Use `create_cust_image_blob.py` a second time to wrap the OTA image into a wired download image. Here, the **wired.ini** file would need to set the "ota" option so the SBL can properly process the data.

   ```
   $ python3 create_cust_image_blob.py -c wired.ini --app_file
   application_ota.bin --output application_ota_wired.bin
   ```

3. Use the `uart_wired_update.py` script to load application_ota_wired.bin into the Apollo4 memory and begin the firmware update process.

   Ambiq SBL performs all the validations of the image in the DTCM before programming to MRAM. This inherently means that the max size for the image that can be downloaded in one go is limited to the available memory to SBL. If you need to download a particularly large image, you can use the **wired_chunk_-size** argument to split the download into multiple pieces. In this case, the bootloader will perform a series of "wired download" operations, each individually verified until the OTA image is fully constructed in the Apollo4 MRAM.

A-SOCAP4-UGGA01EN v1.6
July 2023