USER'S GUIDE

Apollo5 Family and Apollo330 Plus OEM Provisioning Tools

Ultra-low Power Apollo SoC Family

Doc. ID: A-SOCAP5-UGGA03EN-2p0

Document Revision 2.0, July 2026

Legal Information and Disclaimers

AMBIQ MICRO BELIEVES THE INFORMATION IN THIS DOCUMENT IS ACCURATE AT THE TIME OF PUBLICATION. HOWEVER, THE INFORMATION MAY CONTAIN TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. AMBIQ MICRO RESERVES THE RIGHT TO MAKE CORRECTIONS, MODIFICATIONS, ENHANCEMENTS, IMPROVEMENTS, OR OTHER CHANGES TO ITS PRODUCTS (AND DOCUMENTATION) AT ANY TIME WITHOUT NOTICE.

INFORMATION IN THIS DOCUMENT IS PROVIDED SOLELY TO ENABLE THE USE OF AMBIQ MICRO PRODUCTS AND IS PROVIDED "AS IS." NO LICENSE IS GRANTED TO DESIGN OR FABRICATE ANY INTEGRATED CIRCUITS BASED ON THE INFORMATION IN THIS DOCUMENT. THIS DOCUMENT MAY NOT BE USED IN CONNECTION WITH ANY LEGAL ANALYSIS CONCERNING THE AMBIQ MICRO PRODUCTS DESCRIBED HEREIN. AMBIQ MICRO DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT.

AMBIQ MICRO PRODUCTS ARE SOLD SUBJECT TO AMBIQ MICRO'S TERMS AND CONDITIONS OF SALE; SOFTWARE IS PROVIDED PURSUANT TO APPLICABLE LICENSE TERMS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, IS GRANTED UNDER ANY INTELLECTUAL PROPERTY RIGHT OF AMBIQ MICRO OR ANY THIRD PARTY BY THIS DOCUMENT.

CUSTOMERS ARE SOLELY RESPONSIBLE FOR THE DESIGN, VALIDATION, TESTING, AND SECURITY OF THEIR PRODUCTS AND APPLICATIONS, INCLUDING THE SPECIFIC MANNER IN WHICH AMBIQ MICROS PRODUCTS ARE INCORPORATED. "TYPICAL" PARAMETERS ARE FOR REFERENCE ONLY AND IT IS THE CUSTOMER'S RESPONSIBILITY TO VALIDATE "TYPICAL" SPECIFICATIONS FOR THEIR APPLICATION.

AMBIQ MICRO PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN APPLICATIONS WHERE PRODUCT FAILURE COULD RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("HIGH-RISK APPLICATIONS") WITHOUT EXPRESS WRITTEN APPROVAL. IF PRODUCTS ARE USED IN HIGH-RISK APPLICATIONS WITHOUT SUCH APPROVAL, THE CUSTOMER SHALL INDEMNIFY AND HOLD HARMLESS AMBIQ MICRO AND ITS AFFILIATES FROM ALL RESULTING CLAIMS, DAMAGES, COSTS, AND EXPENSES, INCLUDING REASONABLE ATTORNEYS' FEES. IN NO EVENT SHALL AMBIQ MICRO BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES, INCLUDING LOST PROFITS OR DATA, ARISING FROM THE USE OF THIS DOCUMENT OR ANY PRODUCT.

"AMBIQ", "APOLLO", "SPOT", AND COMBINATIONS THEREOF ARE TRADEMARKS OF AMBIQ MICRO, INC. OTHER PRODUCT NAMES USED IN THIS PUBLICATION ARE FOR IDENTIFICATION PURPOSES ONLY AND MAY BE TRADEMARKS OF THEIR RESPECTIVE COMPANIES.

Table of Contents

List of Figures

List of Tables

1. Document Revision History

Table 1: Document Revision History

Rev #	Date	Description
1.0	May 2025	Initial release
2.0	July 2026	▪ Added support for Apollo510 Lite and Apollo330 Plus devices ▪ Updated for SDK 5.2.0 release

2. Reference Documents

Table 2: Reference Documents

Document ID	Description
A-SOCAP5-UGGA04EN	Apollo5 Family and Apollo330 Plus Security User's Guide
A-SOCAP5-UGGA02EN	Apollo5 Family and Apollo330 Plus Secure Update User's Guide
A-SOCAP5-UGGA01EN	Apollo5 Family and Apollo330 Plus MRAM Recovery User's Guide

3. Introduction

This document provides an overview of the OEM provisioning process, initial configuration, and run time updates for the Apollo510, Apollo510L, and Apollo330P Family SoC and the details of the tools required for the same. References to any of the SoCs in Apollo510, Apollo510L, and Apollo330P will by default reference to all the SoCs, unless explicitly stated otherwise.

NOTE

Unless specifically noted, all the tools mentioned below are included in the AmbiqSuite SDK release under directory **/tools/apollo510_scripts/.** As additional Apolloxxx devices are released, each device will have its own dedicated scripts directory under **/tools/**.

3.1 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 – 3.8.10 or later
  - pyserial 3.5 (required for **uart_wired_update**.py and uart_recovery_host.py)
  - cryptography v44.02 (required for encrypted or signed image formats)

The following python libraries are needed only when running the remote signing server example:

- Flask
- Verify that urllib3 version is > 2.0 if OpenSSL version >= 1.1.1

3.2 Terminology

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

Table 3: Terminology

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

4. Keys

The Apollo5 Security infrastructure relies on asymmetric and symmetric keys for a variety of purposes, ranging from creating a Root of Trust, to using asymmetric keys for Authentication, and symmetric keys for various encryption needs.

The scripts and config file paths in this section use **/tools/apollo510_scripts/** as the example SDK directory. For other supported devices, use the corresponding device-specific directory, such as ***/tools/apollo330P_scripts/***.

- **Signing Keys**

The Apollo5 uses Asymmetric PKA for establishing authenticity of loaded images as well as update images.

A Secure Boot 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 INFOC-OTP Root of Trust. Images/certificates are signed using the corresponding Private Keys. Key assets are maintained as Password encrypted .pem files.

NOTE
In cases where the OEM lacks direct access to private keys (e.g., when using a signing server), the AmbiqSuite SDK provides a plugin interface to enable communication with a remote signing server for certificate signing.

- **Encryption Keys**

The Apollo5 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 INFOC-OTP during the OEM provisioning process. Specifically, OEMs program the Apollo5 with a couple of hardware keys Kcp, a Kce, and a bank of additional AES keys (known as the keybank) in the INFOC-OTP.

- **Key Generation**

While the OEM may have their own process to generate the key assets, AmbiqSuite SDK provides simple utilities that can be used to generate them as well.

4.1 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 keys.

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

Upon successful execution, the Key Gen Utility creates the following two folders containing all the required symmetric and asymmetric keys for INFOC-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.

4.2 Signing Server Plugin Support

Ambiq provides tools for generating certificates with customizable signing via plugins, including examples for local and remote signing (with HSM example). Plugins follow a standardized interface for signing operations, allowing users to implement security features tailored to their needs while remaining compatible with future Ambiqsuite SDK updates.

Key Features:

1. Plugin Framework:
   A. Located in **`cert_gen_utils/plugin/signature_interface.py`**.
   B. Default plugin (**LocalInterface**) supports local signing; configurable for remote signing (e.g., via **HTTPInterface**).
   C. Plugins must implement **sign(keyIdx: str, data: bytes**) and can include custom configuration files.
2. Configuration Files:
   A. Includes essential fields like auth-key, signature-plugin-path, and signature- plugin-cfg.
   B. Example entries provided for quick integration.
3. Simulated HSM Example:
   A. Provided in **`cert_gen_utils/plugin_reference/sign_server.py`**.
   B. Enables remote signing via Flask-based server.
   C. Configurable through the **`am_remote_keys_plugin.cfg`** file.
   D. Supports signing and verification operations using RSA encryption.

This framework streamlines secure certificate generation and simplifies plugin customization for advanced use cases. For more information regarding plugin implementation, please review the README found in:

`./oem_tools_pkg/cert_utils/cert_gen_utils/plugin_references/README.txt`

5. OEM Provisioning

OEM provisioning is the process of creating digital security assets in a form that is suitable for the customer's production flow. The Apollo5 provides two different flows for generating the OEM's Security Assets needed to move the device from DM-LCS to Secure-LCS. The first flow creates a plain (unencrypted) data blob.The alternate encrypts the OEM assets using the ICV Key Request/Response from Ambiq that allows the encryption of the Security assets so that the OEMs keys and ROT cannot be compromised. This may be needed if the provisioning tools are being handled by untrusted individuals, if, for example when the products are being manufactured and provisioned by a third party in a factory where the provisioning assets may be handled by non-trusted individuals.

Figure 1 shows the general flow for the plain (unencrypted) OEM security asset generation for the Apollo5 Family.

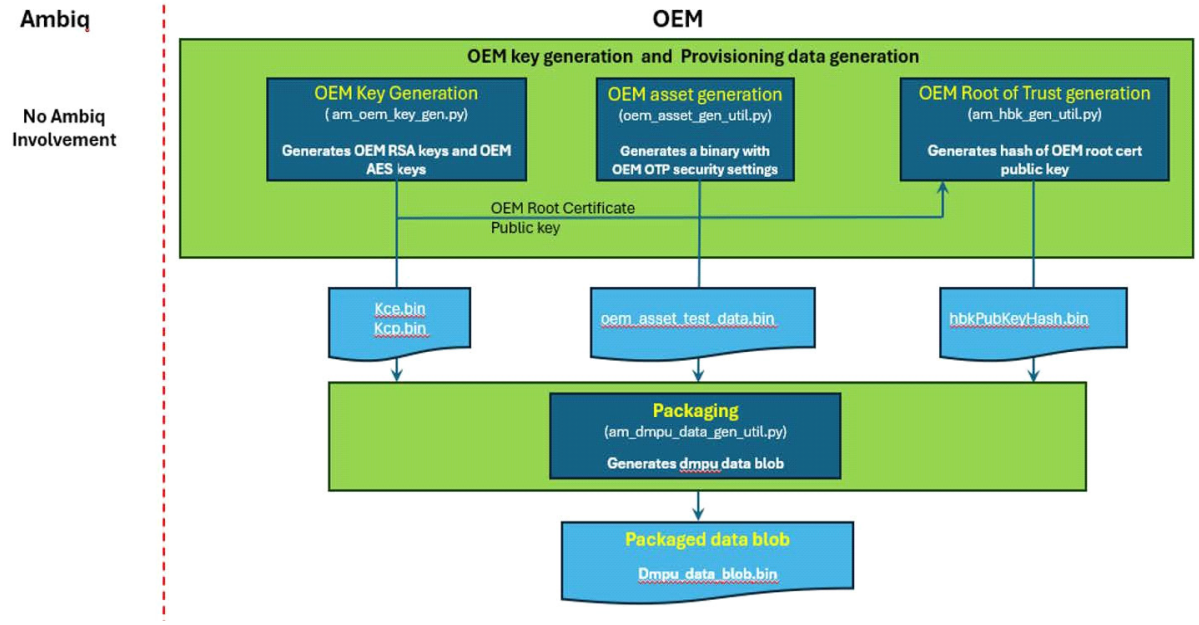

Figure 1. OEM Plain (Unencrypted) Provisioning Data Blob Generation

Figure 2 shows the flow for the generation of the Encrypted OEM security asset generation.

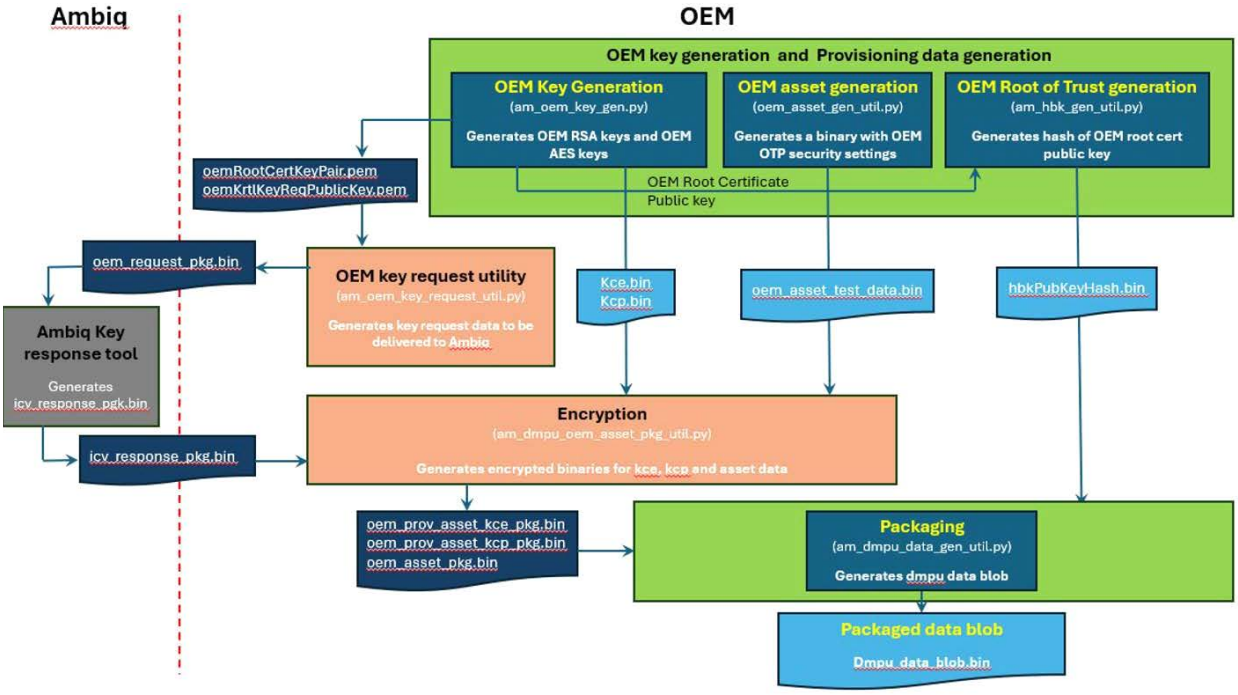

Figure 2. OEM Encrypted Provisioning Data Blob Generation

Ambiq provides the tools for each of these steps in the **/tools/Apollo510_scripts/ oem_tools_pkg** directory. The following subsections will assume this as the base directory for pathnames for the utilities described.

5.1 OEM Key Generation

The Key Gen Utility is used to generate the asymmetric keys required for OEM provisioning at the device manufacturing.

The following python utility is used to generate the keys:

./oem_tools_pkg/am_oem_key_gen_util/am_oem_keys_gen_util.py

5.1.1 Input Parameters

The inputs to the Key Gen Utility are provided in a configuration file passed as a command-line parameter. The inputs specified in the configuration file are described in the example config file located at:

./oem_tools_pkg/am_oem_key_gen_util/oemKeyGenConfig.cfg

5.1.2 Command

The following command runs the Key Gen script:

python am_oem_keys_gen_util.py ./oemKeyGenConfig.cfg

After a successful execution, the Key Gen Utility creates the following two folders containing all the required symmetric and asymmetric keys for INFOC-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 are generated separately by the OEM and input during the OEM asset generation in the next section.

5.2 OEM Asset Gen Utility

The OEM Asset Gen Utility generates an OEM Security binary file used to initialize OEM INFOC-OTP security settings. These settings are specified in the file named **oem_asset_gen.cfg** which is an input to the **oem_asset_gen_util.py** script.

The tool creates a binary data file that is an input to the OEM Provisioning Data Gen Utility discussed in the *Section 5.6: OEM Provisioning Data Gen Utility on page 15.*

5.2.1 Input Parameters

The OEM's INFOC-OTP security setting can be modified in the configuration file:

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

The SDK also contains example files for creating "secure" (Authentication and encryption required on any update files) and "non-secure" (Authentication and encryption not required) OEM Security binary file. These **.cfg** files are contained in the same **/am_config** directory.

- **oem_asset_gen_nonsec.cfg** - config that does not require update files to be encrypted or authenticated. (non-secure)
- **oem_asset_gen_sec.cfg** - config that requires update files to be both encrypted or authenticated. (secure)

5.2.2 Command

The following command is used to run the **oem_asset_gen_util.py** to generate the encrypted OEM assets blob:

Run from:

./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 file **oem_asset_test_data.bin** in the same directory it was run from.

5.3 HBK Gen Utility

The HBK Gen Utility generates the hash of the Root-of-Trust's public key for the OEM (HBK 1).

./oem_tools_pkg/cert_utils/am_hbk_gen/am_hbk_gen_util.py

5.3.1 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 >

5.3.2 Command

Run from:

./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

5.4 OEM Key Request Utility

> **NOTE**
>
> This step is only needed when generating an encrypted OEM security data image. For plain (unencrypted) data this step can be skipped.

The OEM Asset Packaging Utility **am_dmpu_oem_asset_pkg_util.py** is used to encrypt the OEM assets before sending it to the device manufacturing to provision the encrypted assets securely and found in the directory:

./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package

This utility encrypts three plain text assets (Kcp, Kce, and the OEM Security binary file, generated earlier) separately and generates three encrypted assets blobs. The plain text OEM security binary file 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/

5.4.1 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**

5.4.2 Command(s)

The following command will generate the key request binary data:

Run from:

./oem_tools_pkg/oem_asset_prov_utils/oem_key_request

python am_oem_key_request_util.py ./am_config/am_dmpu_oem_key_request.cfg

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

./oem_tools_pkg/oem_asset_prov_utils/oem_key_request/oem_request_pkg.bin

The file is sent to Ambiq that will process it and will return the file **icv_response_pkg.bin.**

5.5 OEM Asset Packaging Utility

> **NOTE**
>
> This step is only needed when generating an encrypted OEM security data image. For plain (unencrypted) data this step can be skipped.

The OEM Asset Packaging Utility **am_dmpu_oem_asset_pkg_util.py** is used to encrypt the OEM assets before sending it to the device manufacturing to provision the encrypted assets securely and found in the directory:

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

This utility encrypts three plain text assets (Kcp, Kce, and the OEM Security binary file, generated earlier) separately and generates three encrypted assets blobs. The plain text OEM security binary file 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/
```

5.5.1 Input Parameters

In the directory:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config
```

Files:

```
oem_asset_enc.cfg
```

```
am_asset_oem_ce.cfg
```

```
am_asset_oem_cp.cfg
```

5.5.2 Command(s)

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

Run from:

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

```
python am_dmpu_oem_asset_pkg_util.py  ./am_config/oem_asset_enc.cfg
```

```
python am_dmpu_oem_asset_pkg_util.py  ./am_config/am_asset_oem_ce.cfg
```

```
python am_dmpu_oem_asset_pkg_util.py  ./am_config/am_asset_oem_cp.cfg
```

The output files containing the encrypted assets will be place in the same directory as follows:

```
oem_asset_pkg.bin oem_prov_asset_kce_pkg.bin oem_prov_asset_kcp_pkg.bin
```

5.6 OEM Provisioning Data Gen Utility

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.

ambiqsuite/tools/apolloxxx_scripts/oem_tools_pkg

5.6.1 Input Parameters

The following config file is used to configure the generation of the OEM provisioning blob:

In directory:

./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/am_config

File: **am_dmpu_data_gen.cfg**

5.6.2 Command

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

Run from:

./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:

./dmpu_prov_data_blob.bin

The SDK provides several example config files for generating various types of data blobs that can be substituted for **am_dmpu_data_gen.cfg**.

- **am_dmpu_data_gen_plain_nonsec.cfg** – This generates an unencrypted OPT data blob (when steps 3.4 and 3.5 were skipped), and Secure Boot is disabled (the image at MAINPOINTER will be run without being authenticated.
- **am_dmpu_data_gen_nonsec.cfg** – This generates an encrypted OPT data blob (when the ICV Key request/response was used). This config also has Secure Boot disabled, when the device transitions to secure.
- **am_dmpu_data_gen_sec.cfg** – This generates an encrypted OPT data blob with Secure Boot enabled. With Secure Boot enabled, the user application will run only when authenticated with **oem_cert_chain**.

5.7 OEM Provisioning Tool (OPT)

The OPT is a Ambiq signed tool which is downloaded and executed on the chip in the OEM's manufacturing facility. It processes the OEM provisioning data and writes the INFOC fields with the selected data and options.

The OPT tool is available at the following location:

./oem_tools_pkg/oem_prov_tool/opt_image_pkg.bin

The tool is downloaded into SRAM at address 0x20030000, and the OEM DMPU asset (encrypted or plain), generated in the previous section, is loaded into SRAM at address 0x20037000. After downloading these 2 blobs, the device is then reset. During the subsequent reboot, the OEM assets get provisioned if both the blobs are authenticated successfully. After a successful OEM provisioning, the device will reset, and the Apollo5 will have been transitioned to Secure LCS.

NOTE

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

5.7.1 Combined OPT Binary

Optionally, both the OPT blobs (**opt_image_pkg.bin** and **dmpu_prov_data_blob.bin**) can be combined into one OPT blob that can be downloaded to 0x20030000.

The following command is used to generate the combined OPT blob:

Run from:

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

```
python am_opt_combined_gen_util.py ./am_config/am_opt_combined_gen.cfg
```

The output of the tool generates as follows:

```
./oem_tools_pkg/oem_asset_prov_utils/oem_asset_package/opt_image_combined.bin
```

6. Customer InfoSpace Provisioning (INFO0)

Customer infospace, also known as INFO0, controls a variety of customer-specific features of the MCU. The following sections provide information on provisioning INFO0 for a particular application.

The scripts for the following examples are also located in **/tools/apollo510_scripts** directory in the AmbiqSuite SDK.

6.1 INFO0 - MRAM and OTP

The Apollo5 device has two "INFO0" spaces, one that is based on MRAM (rewritable) and one that is OTP (one time programmable). Only one of which is active at any one time, providing INFO0 configuration data to the device. Either can be written at any time. Which one is active is determined by the selector register **INFOC_SHDW_TRIM_INFO0_SEL** at (0x400C23FC). See the register documentation for this register and its use.

INFO0-OTP is 256 bytes in size and INFO0-MRAM is 2048 bytes (2K) in size. The first ~160 bytes have Ambiq defined use (that is the same for both Info0-MRAM and Info0-OPT), with the remaining space available to the customer for their use.

The **create_info0.py** script described below always outputs 256B binary file which can be programmed to either the INFO0-OTP or INFO0-MRAM. The customer can modify the script for their use if locations beyond the first 256 bytes are to be used.

6.2 Info0 Generation

The script *create_info0.py* is used to create a binary file which is used to program/update INFO0 (either MRAM or OTP). The user specifies INFO0 parameters either via command-line options, or in a config file, or a combination of the two, with command-line arguments taking precedence. All options can be found in Appendix A. A reduced list of options (minus the MRAM recovery specific options) can be accessed by using the --help flag, the full list can be displayed with the -hx (help extended) flag:

```
python create_info0.py --help
```

While the arguments are listed as optional from the script's execution perspective, various options are required depending on the desired use of the Apollo5 device.

```
$ python create_info0.py --help
usage: create_info0.py [-h][-hx] [--info0Cfg] [--valid {0,1,2}]
       [--version VERSION] [--main MAINPTR]
       [--cchain CERTCHAINPTR][--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 Apollo5 Info0 Blob
positional arguments:
output                    Output filename (without the extension)
optional arguments:
-h, --help                show this help message and exit
```

```
--hx                    show extended help message and exit
--info0Cfg              Relative path to INFO0 configuration file.
--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 = 0x410000)
--cchain CERTCHAINPTR   Certificate Chain location (Default = 0x00000000)
--wTO WIREDTIMEOUT      Wired interface timeout in millisec (default = 2000)
--u0 U0                 UART Config 0 (default = 0x00000000)
--u1 U1                 UART Config 1 (default = 0x00000000)
--u2 U2                 UART Config 2 (default = 0x00000000)
--u3 U3                 UART Config 3 (default = 0x00000000)
--u4 U4                 UART Config 4 (default = 0x00000000)
--u5 U5                 UART Config 5 (default = 0x00000000)
--sdcert SDCERT         Secure Debug Cert Address (default = 0x7ff400)
--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:

To create INFO0 image with Wired UART set on pins 53 (Tx) and 55 (Rx) with baud 115200 (0x1C200), and the timeout of 5 Sec. running the Main image (if non-secure boot) at 0x410000, and the SD Cert location set to 0x7FF400 for Apollo510 and 0x5FF400 for Apollo510L and Apollo330P. If configured for Secure Boot, the default OEM cert chain is located at 0x4C0000.

```
python ./create_info0.py --valid 1 --u0 0x1C200C0 --u1 0xFFFF3537 --u2 0x4
--u3 0x4 --u4 0x0 --u5 0x0 --main 0x410000 --version 1 --wTO 5000 --sdcert 0x7FF400 --
cchain 0x4c0000 info0
```

This will generate an **info0.bin**, which can then be used to program to INFO0 in the Apollo5 device (either INFO0-MRAM or INFO0-OTP).

6.3 Info0 Programming

The generated info0.bin can be programmed directly using a debugger (Jlink script supplied), programmatically using HAL functions, or via the Wired Update process supported by the SBL.

AmbiqSuite SDK provides two sample scripts, **jlink-prog-info0.txt** and **jlink- prog-info0-otp.txt** for the direct programming of Info0 (MRAM and OTP respectively) using a debugger.

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(for Info0-MRAM)
```

or

```
JLink.exe -CommanderScript jlink-prog-info0-otp.txt(for Info0-OTP)
```

Alternatively, INFO0 can be programmed using the SBL assisted update flow, by generating an Info0 update image (see *Section 10.3.3: Info0 Update Images on page 37*) and then using one of the supported methods to download the update image and initiate an update (see *Section 11: Downloading Images and Initiating Updates on page 38*).

7. OEM Image Certificate Generation

Provisioning of the Apollo5 Family in the DM LCS to support Secure Boot depends upon a "chain" of public key certificates as shown in Figure 3. This method provides flexibility and additional security in case the content changes or a certificate in the chain is compromised.

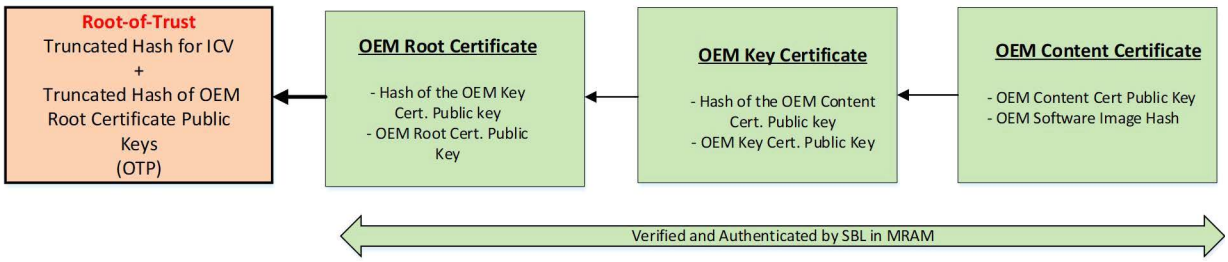

Figure 3. OEM Certificate Generation

7.1 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 INFOC-OTP), as the OEM Root Certificate needs to be authenticated using the RoT. When in DM LCS, before the ROT is programmed, the OEM Root Certificate authentication is bypassed, and accepted as valid, but the remaining certificates in the chain continue to be validated.

7.2 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.

7.3 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
```

7.3.1 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
```

7.3.2 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

7.4 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

7.4.1 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

7.4.2 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

7.5 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

7.5.1 Input Parameters

The input 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. 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 is 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 (e.g., execute only), as it will otherwise fail because of read operation within the image.
- **ExtendedFormat** - Not Supported. Must be set to 0.

7.5.2 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

8. OEM Debug Certificate Generation

A debug certificate(s) is used to enable debugging or transitioning 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 4. 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.

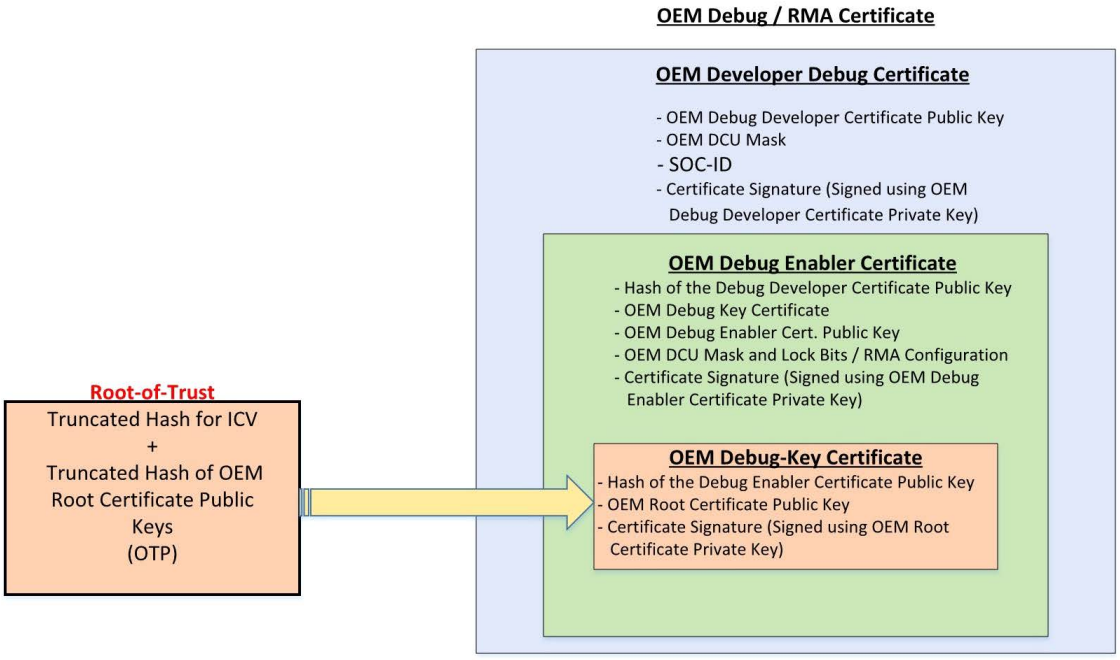

Figure 4. OEM Debug Certificates

8.1 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.

In the directory:

`./oem_tools_pkg/cert_utils/cert_gen_utils/`

Script: **am_cert_key_util.py**

8.1.1 Input Parameters

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

In directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/am_config

File: **am_oem_dbg_key_cert.cfg**

8.1.2 Command

The following command generates the OEM Debug-Key Certificate:

Run in directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/

python amcert_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:

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/am_cert_key_util_output

Files:

debug_oem_key_cert.bin

debug_oem_key_cert_Cert.txt

8.2 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.

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/

Script: **am_cert_dbg_enabler_util.py**

8.2.1 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 example configuration file.

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/am_config

File: **am_oem_dbg_enabler_cert.cfg**

8.2.2 Command

The following command generates the OEM Debug Enabler Certificate:

Run in the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/

python am_cert_dbg_enabler_util.py./am_config/am_oem_dbg_enabler_cert.cfg

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

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/am_debug_cert_output

File: **oem_dbg_cert_enabler_pkg.bin**

8.3 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).

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/

Script: **am_cert_dbg_developer_util.py**

8.3.1 Input Parameters

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

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/am_config

File: **am_oem_dbg_developer_cert.cfg**

8.3.2 Command

The following command generates the OEM Developer Certificate:

Run in the directory:

./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:

In the directory:

./oem_tools_pkg/cert_utils/cert_gen_utils/am_debug_cert_output

Files:

oem_dbg_cert_developer_pkg.bin

oemDeveloperCert.h

9. Transition to RMA LCS

Transition to RMA LCS is required when devices need to be returned to the manufacturer due to device failure. This process involves invalidating the OEM's security (if the device is in Secure LCS) assets (keys, RoT, etc.) which opens the device to perform fault analysis. This process requires signed certificates from both Ambiq and OEM (RMA Certificate) to invalidate the assets.

Note: The RMA certificate uses the same infrastructure and storage location as a standard secure debug certificate. As a result:

- The RMA certificate is installed at the same address as the secure debug certificate, and
- A Secure Debug certificate and an RMA certificate cannot coexist on the device at the same time

9.1 Transition from DM LCS to RMA LCS

A device in DM LCS can be transitioned to RMA LCS using two certificates

- OEM DM-RMA certificate
- Ambiq DM-RMA certificate

While the device is in DM LCS, access to the debug port via SWD remains available. This allows the RMA certificates to be written directly to the secure debug certificate location in MRAM specified by:

- INFO0->INFO0_SBR_SDCERT_ADDR (0x42000058)

9.1.1 DM-RMA LCS Debug Cert Gen

The DM-RMA LCS Debug certificate generation uses the same scripts as the debug certificate described in Sections 8.1 - 8.3.

9.1.2 Input Parameters

The input can be given in the command line options. They are also listed in the example configuration listed below. The LCS option in the enabler RMA certificate configuration file needs to match the current LCS state of the device, which in this case will be "1".

In the directory:

oem_tools_pkg/cert_utils/cert_gen_utils/am_config

Files:

am_oem_dbg_key_cert.cfg;

am_oem_dbg_enabler_cert_rma.cfg;

am_oem_dbg_developer_cert_rma.cfg

9.1.3 Command

Running the following commands will generate the DM-RMA LCS debug certificate.

Run in the directory:

oem_tools_pkg/cert_utils/cert_gen_utils

python am_cert_key_util.py am_config/am_oem_dbg_key_cert.cfg

python am_cert_dbg_enabler_util.py am_config/am_oem_dbg_enabler_cert_rma_LCS1.cfg

python am_cert_dbg_developer_util.py am_config/am_oem_dbg_developer_cert_rma.cfg

Output File:

oem_dbg_cert_developer_pkg_rma.bin

9.2 Transition from Secure LCS to RMA LCS

For a device in Secure LCS, the transition to RMA LCS can be performed through the SBL wired update mechanism or directly loading the RMA CERT (if debugger access is enabled).

The customer is expected to perform the following steps to get the device in partial RMA.

- Load and install the OEM Secure-RMA certificate either using the debugger or SBL's wired update mechanism (Refer to Section 13: Wired Download Procedure on page 42). This will put the device in a partial RMA state invalidating the customer's security assets.
- Enable the option to install Ambiq Secure-RMA certificate.
  - Enable debugger (Either as default or in the main image)
  - Valid INFO0 and INFO0->RMAOVERRIDE enabled(=0x2), allows Ambiq to use SBL wired download once the device has already gone through the OEM's RMA processing
- Provide the SOC ID (if in Secure LCS)

NOTE

If debugger access is enabled by a Secure Debug certificate, then the OEM RMA certificate should be delivered via wired update. Loading the OEM RMA certificate over the debugger that is enabled by SD certificate will overwrite the SD cert consequently disabling the debugger access.

9.2.1 Secure-RMA LCS Debug Cert Gen

The Secure-RMA LCS Debug certificate generation uses the same scripts as the debug certificate described in Sections 8.1 - 8.3.

9.2.2 Input Parameters

The input can be given in the command line options. They are also listed in the example configuration listed below:

In the directory:

oem_tools_pkg/cert_utils/cert_gen_utils/am_config

Files:

am_oem_dbg_key_cert.cfg

am_oem_dbg_enabler_cert_rma.cfg

am_oem_dbg_developer_cert_rma.cfg

9.2.3 Command

Running the following commands will generate the Secure-RMA LCA debug certificate.

Run in the directory:

oem_tools_pkg/cert_utils/cert_gen_utils

python am_cert_key_util.py am_config/am_oem_dbg_key_cert.cfg;

```
python am_cert_dbg_enabler_util.py am_config/am_oem_dbg_enabler_cert_rma.cfg;
python am_cert_dbg_developer_util.py am_config/am_oem_dbg_developer_cert_rma.cfg
```

Output File:

oem_dbg_cert_developer_pkg_rma.bin

10. Image Generation for Apollo510/510L and Apollo330 SBL

The Apollo5's boot ROM and SBL (Secure Bootloader) are capable of performing a wide variety of update 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.

10.1 Overview of Image Types

10.1.1 Firmware

Raw application binaries are the most basic image format for Apollo5. This is simply a compiled application in the standard Arm binary format and can be loaded directly into MRAM using any SWD programming tool (including the JLINK device on the Apollo5 evaluation kits). All later firmware images will be derived from this basic image.

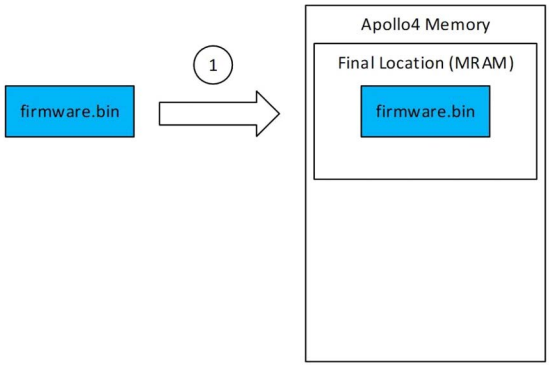

Figure 5. Firmware

10.1.2 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.

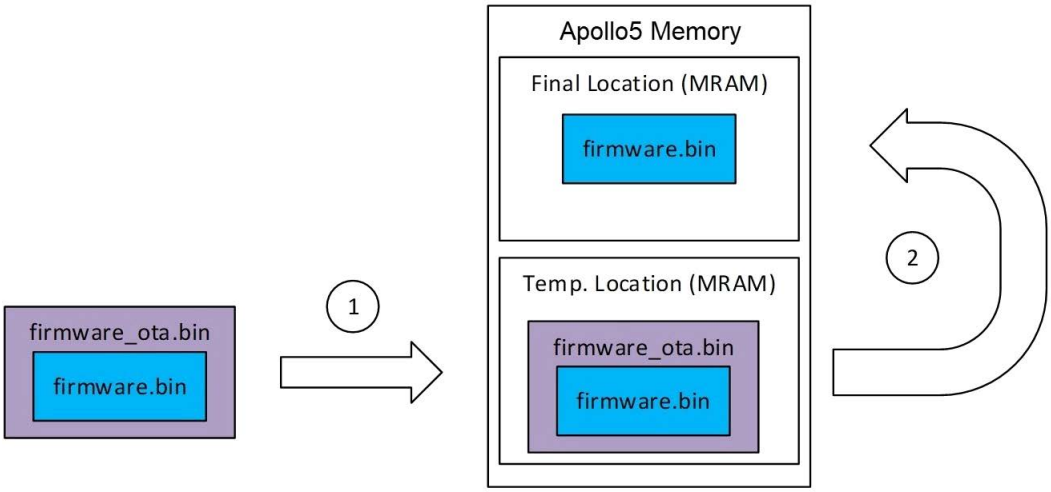

Figure 6. Firmware OTA

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 then triggering the SBL to perform the second step of the upgrade. See the **tools/apollo510_scripts** directory in AmbiqSuite to find these scripts. The scripts also serve as a reference for the process an application would need to follow when performing 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.

10.1.3 Wired Download

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

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 Apollo5 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 Apollo5 device. The SBL will read the wired download data directly over the UART interface, potentially decrypting it or authenticating using an RSA signature, and program the data to its final location in MRAM. No temporary MRAM is required for this process.

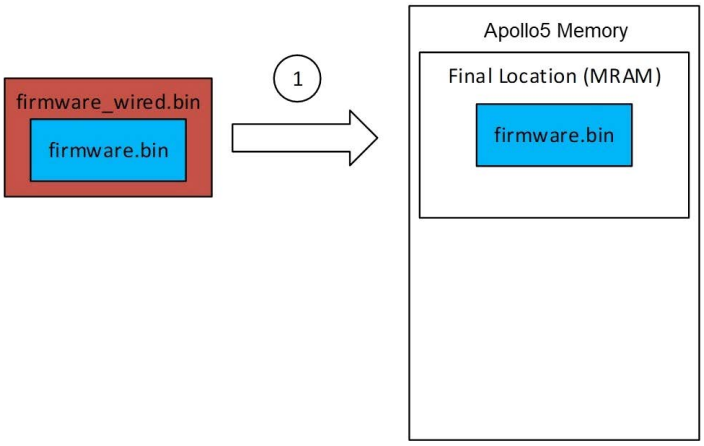

Figure 7. Wired Download

10.1.4 Wired OTA

The OTA and Wired Download formats from the previous sections can be used simultaneously in 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 both 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 the current known good image until the update is fully decrypted and authenticated.

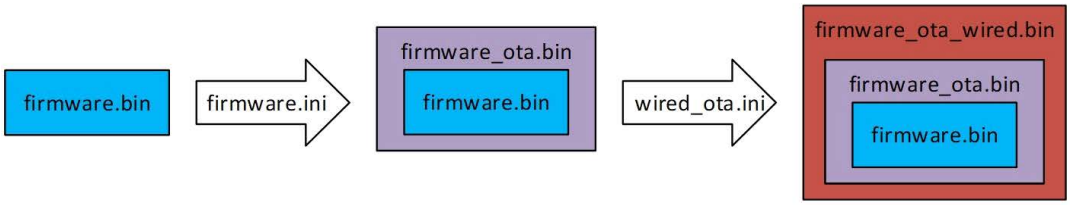

Figure 8. Wired OTA

The update process with a Wired OTA image is similar to the standard OTA process, but the 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 Apollo5 will reboot and perform step 2, where it processes the OTA image and programs the device firmware to its final location.

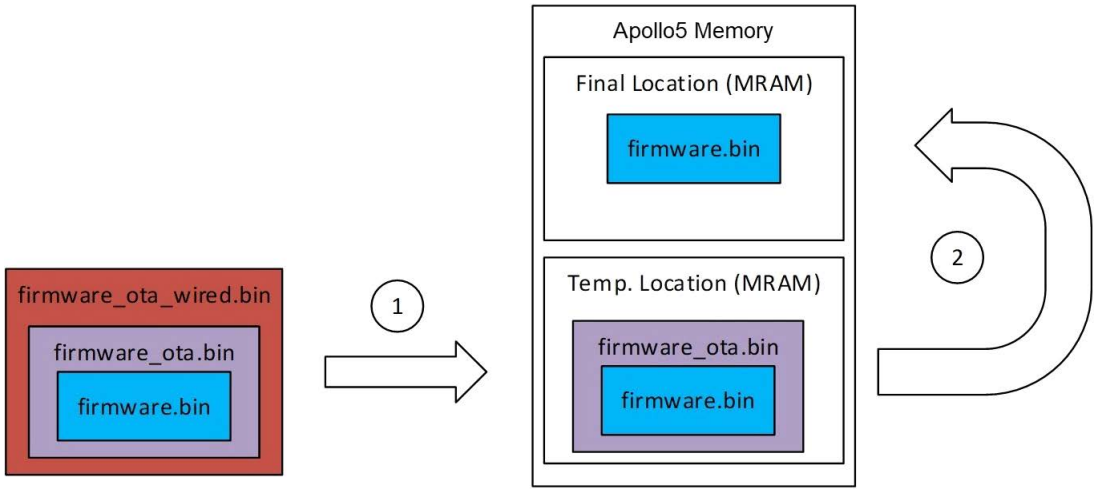

Figure 9. Wired OTA - Step 2

10.1.5 Summary

Figure 10 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.

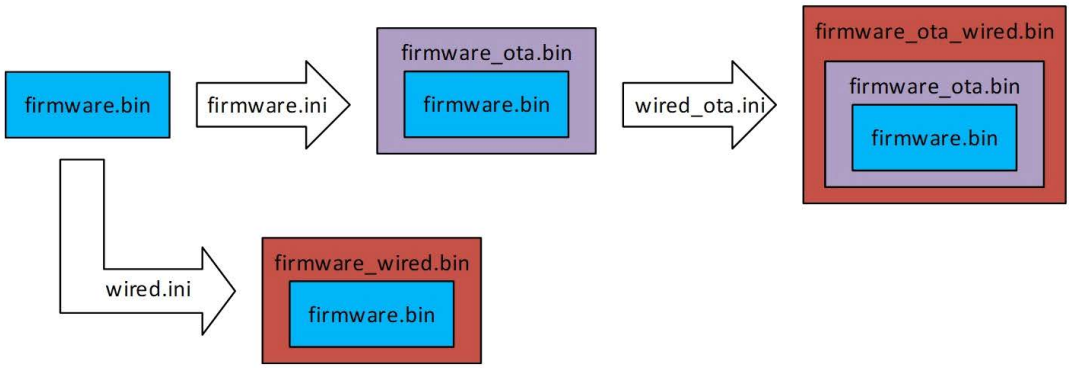

Figure 10. Relationship Summary

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

10.2 Image Generation Scripts

The Apollo5's SBL can perform a wide variety of functions, each requires a binary image with a specific format, which is then loaded into the device over one of the support interfaces. The create_cust_image_blob.py script described in this section can generates these image formats, including the following:

- Secure and non-secure Firmware OTA
- Wired Download
- Certificate

- Chain Update
- Key
- Revocation
- INFO0
- Update
- OEM Application Recovery

10.2.1 Basic Script Usage

Most image formats for the Apollo5 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. The option required will vary depending on the image type being created. 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/ apollo510_scripts/ examples** directory of AmbiqSuite SDK.

10.2.2 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 hexadecimal
# "0x" notation.
#
# To re-generate this file using all default values, run
# "create_cust_image_blob.py --create-config"
#
#*******************************************************************************
[Settings]
chip = apollo510
app_file = hello_world.bin
# Location where the image should be installed load_address = 0x410000
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.

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

Equivalently, this configuration can also be expressed on the command line as fol- lows.

```
$ python ./create_cust_image_blob.py --chip apollo510 --bin hello_world.bin --load-address 0x410000 --enc-algo 0 --kek-index 0x80 --auth_algo 0 --auth-key 0x2 --image-type firmware
```

10.2.3 Universal Security Options

The Apollo5 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 3072 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 rep- resents 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 INFOC-OTP key-bank keys, with each index representing a 128b AES key.
- **key_table**: Path to a configuration file describing the encryption keys used with the Apollo5.

Note: This 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 INFOC-OTP keys programmed in the chips, and are referenced using pointers to binary key and key bank 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/oemRootCertKeyPair.pem
passfile = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/pwdOemRootCertKey_Rsa.txt

[Key Cert Key]
index = 0x1
format = pem
filename = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/oemKeyCertKeyPair.pem
passfile = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeyspwdOemKeyCertKey_Rsa.txt

[Content Cert Key]
index = 0x2
format = pem
filename = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/oemContentCertKeyPair.pem
passfile = oem_tools_pkg/am_oem_key_gen_util/oemRSAKeys/pwdOemContentCertKey_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
```

10.3 Generating Specific Image Types

10.3.1 Firmware OTA Images

Firmware OTA images contain firmware and metadata to be used by the boot- loader 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 cer-
- tificate 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.

10.3.2 Wired Download Images

Wired download images can contain any data blob and instructions for programming to a specific location. The bootloader will read the image and program the blob to the specified 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 Apollo5 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 Apollo5 should download in a single pass. See wired OTA description in *Section 12.1: Upgrading Multiple Images in One Step on page 41*.

10.3.3 Info0 Update Images

This image type can be used to update the INFO0 data area in the Apollo5 device. INFO0 is used to contain certain non-volatile Apollo5 device settings. For more information, see the **create_info0.py** script documentation in *Section 6.2: 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.
- **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.

10.3.4 OEM Certificate Chain Update

OEM Chain Update images are used to make updates to the certificate chain used by the Apollo5 SBL. The root certificate, key certificate, and content certificate all get changed using this process. This option will be primarily used to increment the certificate version to ensure images cannot be rolled back to older certificates.

- **image_type**: set to "oem_chain"
- **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.

10.3.5 Key Revocation Images

The Apollo5 "keybank" keys (as discussed in *Section 4: Keys on page 9*) are programmed into INFOC-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 provide 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 in the keybank. Setting a bit revokes the key with the same number (for example, setting bit 0 revokes keybank AES key 0).

11. Downloading Images and Initiating Updates

As described in the previous chapter, the Apollo5 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 Apollo5 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.

11.1 SWD Download Using JLINK

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

AmbiqSuite includes JLINK scripts that demonstrate how the JLINK can be used to program a specific memory location in the Apollo5 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.

This script can be run by the JLINK command line utility the following command (for Windows):

```
JLink.exe -CommanderScript jlink-ota.txt
```

- **SBL Update**

Jlink-ota-bootrom.txt: Installs a replacement Secure Bootloader (SBL) done through the bootrom in the Apollo5 device in a similar manner, but uses the jlink script jlink-ota-bootrom.txt to perform a SBL-OTA update.

This script can be run by the JLINK command line utility the following com- mand (for Windows):

```
JLink.exe -CommanderScript jlink-ota-bootrom.txt
```

11.2 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.

11.3 Over-the-Air Updates

The SBL can also be used to support over-the-air updates enabled by user firmware. In this case, the user firmware will communicate with a boot host and download an OTA image (described in the image formats section earlier). Once the download is completed, the user application can program the update blob information by programming the OTA infrastructure (see *Apollo5 Family Secure Update User's Guide*), and initiate a device reset. The SBL takes over on the next reboot and validates the downloaded image, and performs a firmware upgrade

The AmbiqSuite SDK provides an example of an OTA firmware update over BLE through the "AMOTA" app for the Apollo5. 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/Apollo5_amota/scripts** can be used to generate an OTA blob compatible with AMOTA. Most of the optional parameters are not relevant for the Apollo5.

Example usage:

```
python  ota_binary_converter.py --appbin main_ota.bin -o main_ota_amota
```

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.

12. UART Wired Update

For UART based wired update to work, the device needs to be provisioned to allow UART wired update through INFOC-OTP and INFO0 settings (see *Apollo5 Family Security User's Guide*). The SBL will get into update mode in one of the two cases:

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

The host needs to be connected to the UART's pins as configured in Info0's UART configurations, and the host initiates the communication within a short time window, which is also configured in Info0.

The script **uart_wired_update.py** is designed to emulate the host side functions when using the UART as the wired interface. It can be used during development to test the UART wired update features, and to program the Apollo5 device while the debugger is disabled. Usage information for **uart_wired_update.py** appears below:

```
$ python uart_wired_update.py --help

\ambiqsuite\tools\apollo510_scripts>python uart_wired_update.py -h

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 Apollo5

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.

```
python  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.

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

12.1 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.

The example below shows upgrading isolated data segments and main image (all considered non-secure main images generated as in *Section 8.1.1: Input Parameters on page 24*) together using **uart_wired_update.py**:

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

```
python uart_wired_update.py  -b 115200 COM<X>  -f img1_nonsecure_wire.bin  --r 0
```

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

```
python uart_wired_update.py -b 115200 COM<X> -f img2_nonsecure_wire.bin --r 0-o
0xFFFFFFFF
```

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

```
python  uart_wired_update.py  -b 115200 COM<X>  -f img3_nonsecure_wire.bin --r 1-o
0xFFFFFFFF
```

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

SBL uses DTCM memory 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 specifying the maximum number of bytes available in the Apollo5 MCU (the maximum block size). The **create_cust_image_blob.py** script will then 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 Apollo5 device using the **uart_wired_update.py** script as normal. The script will detect that the wired update image is composed of multiple pieces and performs the download accordingly.

13. Wired Download Procedure

The standard use of wired download images is to allow a binary image to be programmed to a device using a method other than direct programming using the 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 Apollo5 memory using the wired download option is as follows:

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

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

Use the resulting **application_wired.bin** file is then transferred using the **uart_wired_update.py** script to load the application into the Apollo5 memory. This will place a copy of **application.bin** directly into the Apollo5 memory at the address specified in wired.ini configuration file.

13.1 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 Apollo5 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 con- tent 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.

   ```
   python 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.

   ```
   python 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 Apollo5 memory and begin the firmware update process.

Ambiq SBL performs all the validations of the image in the DTCM memory 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 Apollo5 MRAM.

14. Appendix A: create_info0.py Options

The **create_info0.py** script is driven by command line arguments and/or config file (.ini) file keyword options. There is an extensive list of options. The table below lists all the options that can be supplied to the **create_info0.py** script.

The **.ini** keywords are used in the config file to specify the various options are based register/ field names defined the Info0 Register documents (minus the **INFO0**, or **INFO0_OTP** prefix). Details of each field and its use can be found in the register documentation.

Fields with a * are subfields of the previous full 32-bit word field that precedes the * grouping. The options can be specified either as the single full 32-bit word value or using the individual sub fields of the word. The .ini file is processed in sequential order and the last processed value for a field/subfield will take precedence (if there is an overlap or duplication). Command line options specified will take precedence over the same options specified in the config file.

Table 4: create_info0.py Options

Cmd Line Args	.ini keyword (config files)	Values (32-bit unless specified)[a]
--valid	VALID	1 = Valid (default) (valid sig w/ data) 2 = Invalid (invalid sig, data all 0s)
--version	SECURITY_VERSION_VERSION	
--main	MAINPTR_ADDRESS	Default = 0x00410000
--cchain	CERTCHAINPTR_ADDRESS	
--wTO	WIRED_TIMEOUT_TIMEOUT	Default = 2000 (millisecond)
--u0	SECURITY_WIRED_IFC_CFG0	
--u1	SECURITY_WIRED_IFC_CFG1	
--u2	SECURITY_WIRED_IFC_CFG2	
--u3	SECURITY_WIRED_IFC_CFG3	
--u4	SECURITY_WIRED_IFC_CFG4	
--u5	SECURITY_WIRED_IFC_CFG5	
--sdcert	SBR_SDCERT	Default = 0x007FF400
--rma	SECURITY_RMAOVERRIDE_OVERRIDE	2=enabled, 5=disabled (default = 2)
--sresv	SECURITY_SRAM_RESV_SRAM_RESV	
--rcvyCtl	MRAM_RCVY_CTRL	
--rcvyEN *	MRAM_RCVY_CTRL_MRAM_RCVY_EN	
--progGPIO *	MRAM_RCVY_CTRL_GPIO_RCVY_INPROGRESS	
--emmcPart *	MRAM_RCVY_CTRL_EMMC_PARTITION	
--wdtEn *	MRAM_RCVY_CTRL_WD_EN	

Table 4: create_info0.py Options

Cmd Line Args	.ini keyword (config files)	Values (32-bit unless specified)[a]
--rcvyPol *	MRAM_RCVY_CTRL_GPIO_CTRL_POL	
--rcvyPin *	MRAM_RCVY_CTRL_GPIO_CTRL_PIN	
--rcvyRbt *	MRAM_RCVY_CTRL_APP_RCVY_REBOOT	
--rcvyTyp *	MRAM_RCVY_CTRL_NV_RCVY_TYPE	
--rcvyModNum *	MRAM_RCVY_CTRL_NV_MODULE_NUM	
--rcvyWire *	MRAM_RCVY_CTRL_WIRED_RCVY_EN	
--rcvyAppEN*	MRAM_RCVY_CTRL_APP_RCVY_EN	
--meta	NV_METADATA_OFFSET_META_OFFSET	
--pwrRstCfg	NV_PWR_RESET_CFG	
--jedecRst *	NV_PWR_RESET_CFG_JEDEC_RESET	
--pwrRstPol *	NV_PWR_RESET_CFG_RESET_POL	
--pwrPol *	NV_PWR_RESET_CFG_PWR_POL	
--pwrDlyUnt *	NV_PWR_RESET_CFG_PWR_DLY_UNITS	
--pwrDly *	NV_PWR_RESET_CFG_RESET_DLY	
--nvPin	NV_PIN_NUMS	
--nvCEPin *	NV_PIN_NUMS_CE_PIN	
--nvRstPin *	NV_PIN_NUMS_RESET_PIN	
--nvPwrPin *	NV_PIN_NUMS_PWR_PIN	
--cmdPinCfg	NV_CE_CMD_PINCFG_CE_CMD_PINCGF	
--clkPinCfg *	NV_CLK_PINCFG_CLK_PINCGF	
--dataPinCfg *	NV_DATA_PINCFG_DATA_PINCFG	
--dqsPinCfg *	NV_DQS_PINCFG_DQS_PINCGF	
--nv0	NV_CONFIG0_CONFIG0	
--nv1	NV_CONFIG0_CONFIG1	
--nv2	NV_CONFIG0_CONFIG2	
--nv3	NV_CONFIG0_CONFIG3	
--nvOpt	NV_OPTIONS	
--emmcWidth *	NV_OPTIONS_EMMC_BUS_WIDTH	
--emmcVolt *	NV_OPTIONS_EMMC_VOLTAGE	
--mspiPadSet *	NV_OPTIONS_MSPI_PADSET1	
--mspiD4Clk *	NV_OPTIONS_MSPI_D4CLK	
--mspiRdClkSel *	NV_OPTIONS_MSPI_READ_CLKSEL	

Table 4: create_info0.py Options

Cmd Line Args	.ini keyword (config files)	Values (32-bit unless specified)[a]
--mspiWidth *	NV_OPTIONS_MSPI_WIDTHS	
--mspiPreClkSel *	NV_OPTIONS_MSPI_PRECMD_CLKSEL	
--mspiPreCtl *	NV_OPTIONS_MSPI_PRECMD_CTRL	
--mspiRdCmd *	NV_OPTIONS_MSPI_READCMD	
--preCMDs	NV_MSPI_PRECMDS	
--preCMD1 *	NV_MSPI_PRECMDS_PCMD0	
--preCMD2 *	NV_MSPI_PRECMDS_PCMD1	
--preCMD3 *	NV_MSPI_PRECMDS_PCMD2	
--preCMD4 *	NV_MSPI_PRECMDS_PCMD3	
--retryWD	MRAM_RCV_RETRIES_TIMES	
--maxRetry *	MRAM_RCV_RETRIES_TIMES_MAX_RETRIES	
--minRetryTm *	MRAM_RCV_RETRIES_TIMES_MIN_RETRY_TIME	
--maxRetryTm *	MRAM_RCV_RETRIES_TIMES_MAX_RETRY_TIME	
--info0Cfg	INFO0CFG	File/pathname to Info0 Config file
--loglevel	LOG_LEVEL	0-5

a. All fields default to 0x0 unless specified.

Ambiq Micro, Inc.
6500 River Place Boulevard, Building 7,
Suite 200, Austin, TX 78730-1156
www.ambiq.com/
sales@ambiq.com
https://support.ambiq.com
+1 (512) 879-2850

A-SOCAP5-UGGA03EN-2p0
Version 2.0
July 2026