



Raptor Lake Signing and Manifesting User Guide

User Guide - NDA

Revision 1.0

July 2022

Intel Confidential



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: <http://www.intel.com/design/literature.htm>

*Other names and brands may be claimed as the property of others.

Copyright © 2022, Intel Corporation. All rights reserved.

Contents

1	Overview	6
1.1	Tools Used in This Document	6
1.2	Terminology	6
1.3	Pre-Requisites	7
2	Introduction	8
2.1	Why is signing important?	8
2.2	Who performs the signing?	8
2.3	When is signing performed?	8
3	Theory of Signing	9
3.1	Cryptography Basics	9
3.2	Key Security	9
3.3	Signed Components and Their Structure	10
3.4	OEM Key Manifest (OEM KM)	11
3.5	Opting out of the OEM KM	12
3.6	Stitching a Flashable Image	12
3.7	IP Loading	13
3.7.1	Boot Flow Order	13
3.7.2	OEM KM Precedence	14
3.7.3	Signature Authentication during Boot	15
4	What Can Be OEM Signed	17
5	How to Sign	18
5.1	High Level Signing of OEM Components	18
5.2	Quick List of Signing Commands	18
5.3	Extended Signing Commands, Detailed Instructions and MEU Abilities	21
5.3.1	Additional ways to generate public key hash	21
5.3.2	Versioning of Signed Components	22
5.4	Intel® Manifest Extension Utility (Intel® MEU)	24
5.4.1	Usage	24
5.4.2	Examples	25
6	Intel® MFIT	35
6.1	Descriptor Signing	35
6.2	Signing components added to Intel® MFIT	36
6.3	Intel® MFIT Manifest Version Validation	37
§	37	
7	Production Signing	38
7.1	Production Signing High-Level	38
7.2	Export Manifests	38
7.3	Manifest structures	39
7.3.1	Manifest Header	39
7.3.2	Signed Package Info Extension	41
7.3.3	Metadata extensions	43
7.3.4	OEM Key Manifest	44



	7.4	Import Manifest	45
8		Common Bring Up Issues and Troubleshooting Table.....	46
	8.1	Common Bring Up Issues and Troubleshooting Table.....	46

Revision History

Revision Number	Description	Revision Date
1.0	<ul style="list-style-type: none">No Changes	July 2022
0.8	<ul style="list-style-type: none">Rebranded	November 2021
0.5	<ul style="list-style-type: none">Initial Release	April 2021

1 Overview

This document describes the manifesting and signing of OEM components, enabling them to be included in the IFWI image for Raptor Lake platforms using Intel® CSME 17 FW.

The goal of this guide is to train the user to:

1. Manifest and sign OEM components
2. Include data on all signatures in the IFWI image
3. Build the final flashable production IFWI image
4. Configurations and options available in the signing process

This guide also offers theory and background for signing and IP loading flow.

There may be components mentioned in this document which are not PoR for RPL.

1.1 Tools Used in This Document

The following tools are referenced this document:

- Intel® Flash Image Tool (Intel® FIT): in Intel® CSME FW Kit
- Intel® Manifest Extension Utility (Intel® MEU): in Intel® CSME FW Kit.
- OpenSSL: Open Source

1.2 Terminology

Term	Description
EOM	End of Manufacturing
FW	Firmware
IFWI	Integrated Firmware Image (System FW Image on SPI)
Intel® MEU	Intel® Manifest Extension Utility
Intel® MFIT	Intel® Modular Flash Image Tool
ISH	Integrated Sensor Hub
IUP	Independently Updatable Partition
OEM KM	OEM Key Manifest (containing OEM public key hashes to authenticate OEM signed FW components).

Term	Description
ROT KM	Root of Trust Key Manifest (containing Intel public key hashes to authenticate Intel signed FW components)
RPL	Raptor Lake

1.3 Pre-Requisites

The user should download and install the [Latest Intel® CSME FW kit](#) from Intel's Resource and Design Center (RDC)

The following guides, found in the CSME FW kit, can offer background for processes and tools discussed in this document:

- [RPL Firmware Bring-Up Guide](#): Describes the overall platform bring-up procedure.
- [RPL System Tools User Guide](#): Offers further detail regarding usage of all FW manufacturing tools.



2 Introduction

2.1 Why is signing important?

When a platform boots, it is critical to ensure the FW is loaded from a trusted source.

Signing of FW components ensures that the owner of the component (OEM/Intel) authorizes the loading and running of their component on the platform. This is done by establishing a chain of trust from the hardware of the platform itself, where hardware authenticates a key manifest, and the key manifest is used to then authenticate the FW components.

Platform Chain of trust extended from HW to OEM components



2.2 Who performs the signing?

Intel signs all FW components to be loaded by CSME. OEMs may add or replace capabilities for several components, such as ISH and Audio. In order to load the OEM components and use their capabilities, signing of the component and an OEM KM is required.

If the OEM wishes to only use the Intel provided components, the OEM is not required to sign anything, and OEM KM is not created.

2.3 When is signing performed?

Signing of components and creation and signing of OEM KM, is a step performed in the R&D facilities pre-manufacturing. At the time of manufacturing, the ready signed OEM components and OEM KM are entered into the image creation tool (FIT) and the key used to authenticate the OEM KM will be burned to the fuses. This will be discussed in greater detail below.

3 Theory of Signing

This chapter discusses the theory of signed structures, signing components and how authentication is performed during boot flow. For technical instructions on how to use the tools to sign your components, please refer to chapter 4.

3.1 Cryptography Basics

Signing flow, and establishing a chain of trust, is based on the concepts of cryptography. Two cryptographic functions are used in the process:

1. **Hashing**

A one directional mathematical operation which is simple to calculate, yet computationally difficult to reverse. It will produce completely different outputs even when input data is similar. For RPL, the hashing function used is SHA-384, which is from the SHA2 family of cryptographic functions.

2. **Data Encryption using RSA Algorithm**

Using a private and public key pair which are mathematically linked, data can be encrypted and then decrypted (reverse encryption). The private key is used to encrypt the data, and then public key can be used to decrypt it back to the original source.

In the signing process of components, the data being encrypted is the hash of the original binary component, and the public key is used to decrypt it back to its original format during verification. It is important for the private key to be stored securely, so that only the original body can perform the encryption. Public key is available to the public, since once it is used to decrypt the signature, the output is compared with the binary hash present in the component. They will only match if the public key mathematically corresponds perfectly to the private key used during encryption.

For RPL, the private key size is RSA-3072.

3.2 Key Security

Although the same key may be used for signing each entry in the OEM Key Manifest and the key manifest itself, Intel recommends using separate key pairs for signing each component. Using a single key for signing multiple components poses a level of risk, since if the key is compromised, the entire package is compromised.

Production private keys should always be stored securely and kept secret to provide a robust secure boot flow and firmware load. If the keys escape to 3rd parties, they may be used to create and sign unofficial versions of the binaries which can then be loaded onto the platform.

It is important to allowing restricted/audited access to the keys in order to resign components and build updated images for the platform.

For example, MEU could be run on a secure server which houses the keys or OEMs may use the MEU export function for production signing if MEU does not run on the OEM's signing server (see production signing chapter).

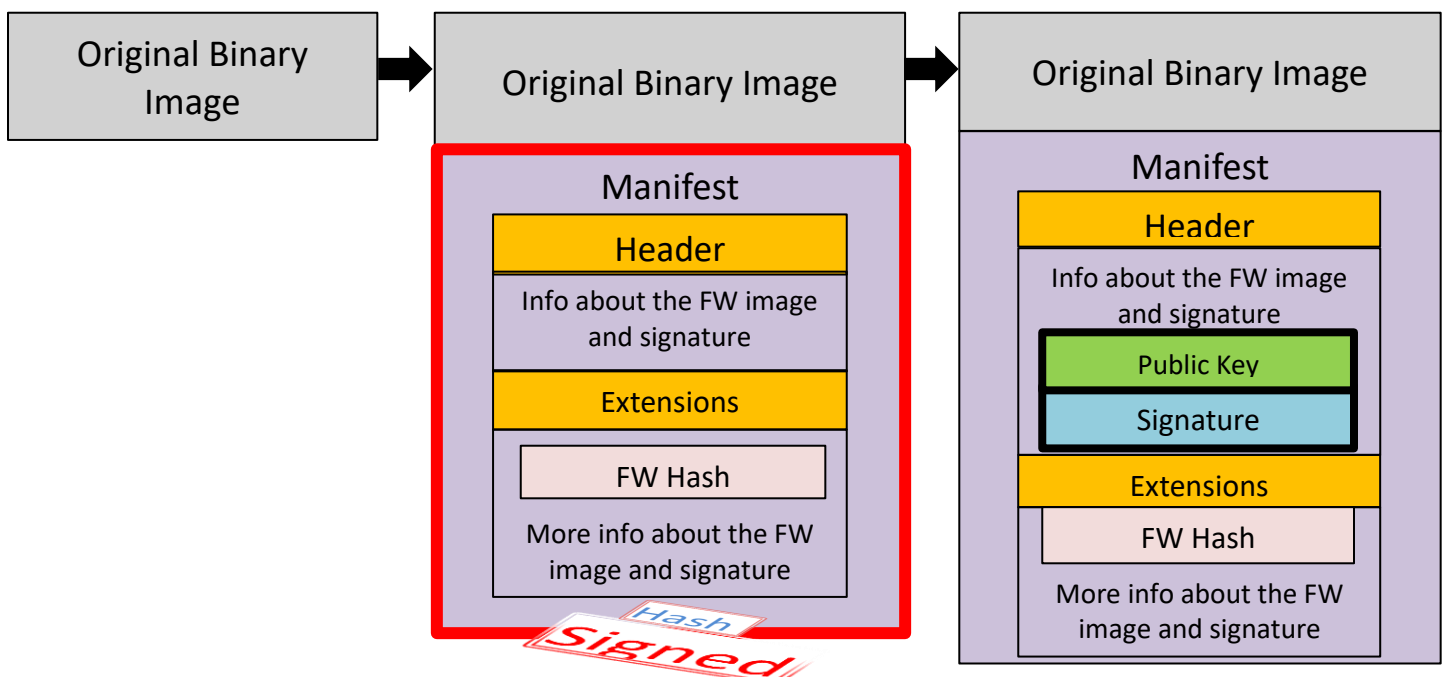
OEMs should manage separate sets of keys for development signing and production signing of images. This will ensure that the OEM KM and components run on production platforms is of production quality.

3.3 Signed Components and Their Structure

The OEM may create and sign ISH to replace the capabilities of the Intel ISH, as well as create and sign an Audio component to extend the Audio capability provided by Intel. Intel iUnit may be re-signed with OEM key, but OEM may not create their own iUnit component. Each one of these is independent. In addition, there are OEM signed binaries that use the signing chain of trust to enable capabilities such as debug tokens and DnX (see corresponding guides in kit collaterals).

Each item that is signed begins with the same structure, a binary, and in the signing flow a manifest is added to it. The manifest is then signed, and the signature and public key are entered into the header of the manifest to create the final signed component binary.

Regardless of the type of binary being signed, all signed components have the same final structure of original binary and manifest, where public key and signature are part of the manifest header. See image:



3.4 OEM Key Manifest (OEM KM)

The OEM Key Manifest plays a central part in the signing mechanism. It lists the public key hashes used for authenticating the OEM-created binaries to be loaded.

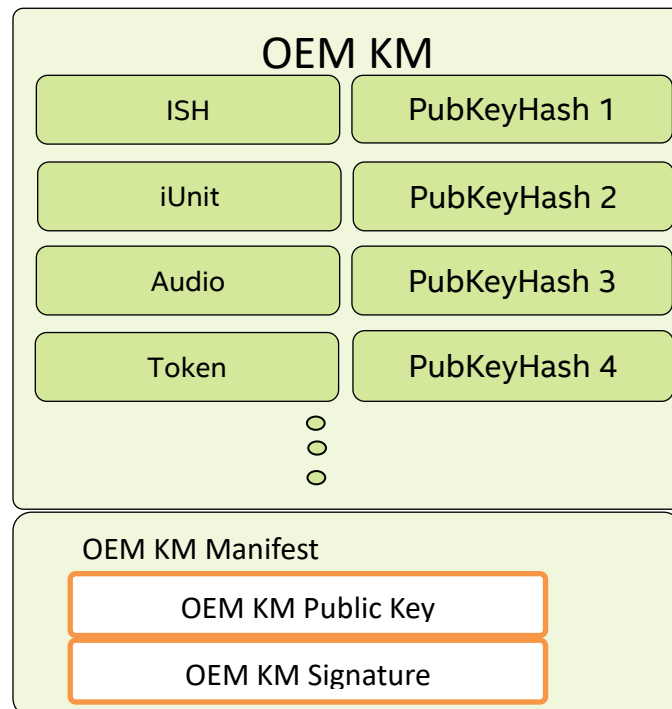
The OEM Key Manifest itself is signed, and its corresponding public key hash is burned into a fuse (OEM FPF) at EOM, so it can never be changed. This creates a secure verification mechanism where firmware verifies that the OEM Key Manifest was signed with a key owned by a trusted owner. Once OEM KM is authenticated, each public key hash stored within the OEM KM is able to authenticate the corresponding FW binary.

Can also add an OEM specific key under "OemAttestationManifest" usage value, which works as a secure storage to hold an OEM key for OEM to use in their own authentication process of their FW.

Important!

Since the hash burned into the platform hardware can never be changed, it is **critical to secure the private key** used to sign the OEM Key Manifest. If at any stage OEM would like to update the image on the platform, the OEM KM for the new image must be signed with the same key used for the original OEM KM.

OEM KM Example:



Note: each component in the OEM KM is independent and can be entered alone, or not entered at all, to OEM KM.

3.5 Opting out of the OEM KM

OEMs who do not wish to utilize the OEM KM, may use Intel signed components authenticated by ROM.

When creating the final flashable image, ensure Intel components will not fail to load due to signature issues by using pre-production Intel signed ISH/Audio/iUnit with pre-production CSME FW & production Intel signed ISH/Audio/iUnit with production CSME FW.

Do not create nor include OEM KM binary into MFIT during image creation. At EOM an FPF will **permanently** be set to indicate that the OEM KM is not present, and that platform image can never be updated with an OEM KM.

Important!

A platform that does not have an OEM KM in the image at the time of EOM, will never be able to load an image containing an OEM KM.

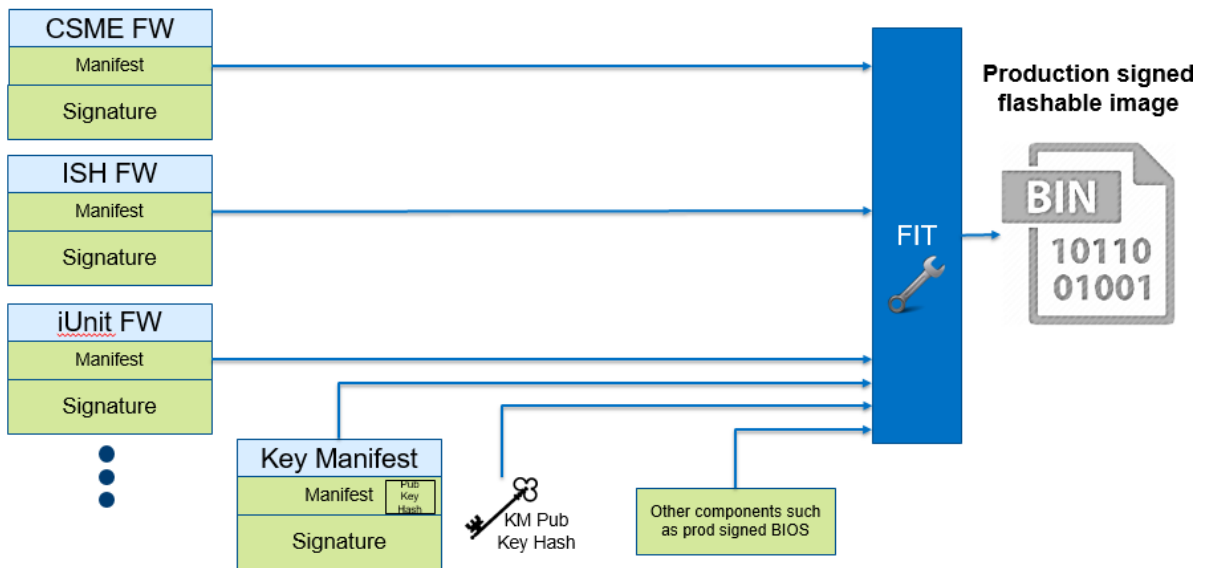
This means that if an OEM chooses not to sign any OEM components at the time of manufacturing, they can never add OEM signed components for that platform.

Intel recommends that OEMs always add an OEM KM, even if they have no use for it at the time the image is manufactured. This can be done by adding an empty OEM KM (with no entries), which holds the spot for an OEM KM which may be added at a later point via FWUpdate in field.

3.6 Stitching a Flashable Image

Intel provides signed components in the kit released to OEMs. As mentioned above, OEMs may create and sign some of their own components. To create the final flashable image, individual components need to be entered into the Flash Imaging Tool (FIT) to stitch the components into the final image.

During image creation using FIT, when OEM signed components are included into the image, the OEM KM and OEM components are added into MFIT in addition to the Intel components. (See System Tools User Guide for more information on MFIT usage.)



3.7 IP Loading

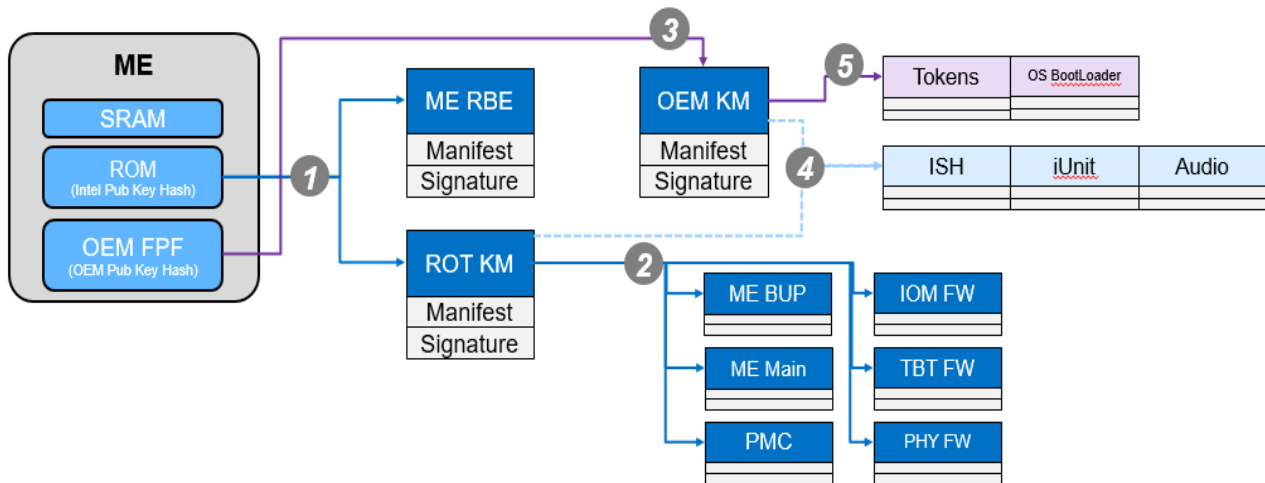
3.7.1 Boot Flow Order

The signing of components is all preparation to be used in authentication of components during boot time.

The boot flow order and establishment of root of trust, is as follows:

1. Using the Intel public key hash stored in ROM HW, RBE and ROT KM are authenticated. (ROT KM holds the public key hashes for the Intel signed components.)
2. Once RBE and ROT KM are authenticated, public key hashes in ROT KM are used to authenticate Intel components; each key authenticates its corresponding component.
3. If an OEM KM is present, RBE will authenticate the OEM KM using the OEM public key hash in the OEM FPF.
4. Once OEM KM is authenticated, the keys inside it are used to authenticate OEM components included in the OEM KM list. If a component can be signed by OEM but is not, RBE authenticates the Intel components against the keys in ROT KM.

5. Lastly, if present, components or capabilities that can only be signed by OEM, are authenticated against the keys in the OEM KM.



3.7.2 OEM KM Precedence

During the authentication process, where relevant, the CSME engine first checks the OEM KM to see if the desired component is listed. If the component is listed in OEM KM, the associated key hash will be used for authenticating the component and determine whether it should load.

If the component is not listed by the OEM as a desired usage in the OEM KM, the CSME engine will look up the key hash in the ROT KM, and determine whether the component can load based on whether it authenticates.

If a public key hash is present in OEM KM, yet it fails to authenticate, CSME **will not** try to authenticate the corresponding Intel components based on ROT KM.

See table below showing the components which can be listed in the OEM KM, and what the precedence is if they are listed.

FW Component	ROT KM	OEM KM	Precedence	ME authentication behavior during FW loading
ME BUP	Y	N	ROT KM	Authenticate using key in ROT KM, if no key or authentication fails, fail to boot.
ME Main	Y	N		
PMC	Y	N		
ISH BUP	Y	N		
Audio (cAVS) Image #1	Y	N		Authenticate using key in ROT KM, if no key or authentication fails, fail to load component.
TCSS and CPU PHYs	Y	N	ROT KM	Authenticate using key in ROT KM, if no key or authentication fails, fail to load component.
PCH PHY (for RPL S segment only)	Y	N	ROT KM	
ISH Main FW	Y	Y	OEM KM then ROT KM	If usage present in OEM KM, authenticate using key in OEM KM. If authenticate fails, fail to load component & exit flow. If usage is not present in OEM KM, authenticate using key in ROT KM. If no key or authenticate fails, fail to load component.
iUnit Boot Loader	Y	Y		
iUnit Main FW	Y	Y		
Audio (cAVS) Image #0	N	Y	OEM KM Only	If key usage marked for component in OEM KM, authenticate using key in OEM KM, if authenticate fails, fail to load component & exit flow.
OS Boot Loader	N	Y		
OS Kernel	N	Y		
OEM Debug Tokens	N	Y		

3.7.3 Signature Authentication during Boot

Every component in the boot flow, Intel and OEM, all go through the same authentication flow to verify the signature of the component. No matter what the component is, RBE, a key manifest or a component such as ISH, the concept is the same.

When platform boots, all that is known to be secure are the public key hashes in the HW (Intel's in ROM, and OEM's in OEM FPF). Every step of the way is started with a public key hash that has been authenticated to be secure, and a component which needs to be authenticated.

The component to be authenticated contains the original binary attached to a manifest which contains the public key and RSA signature.

The following three steps authenticate the binary to be loaded:

1. **Verify Public Key**

Public key found in the manifest header is hashed and compared with the already verified public key hash used to authenticate the component. For example:

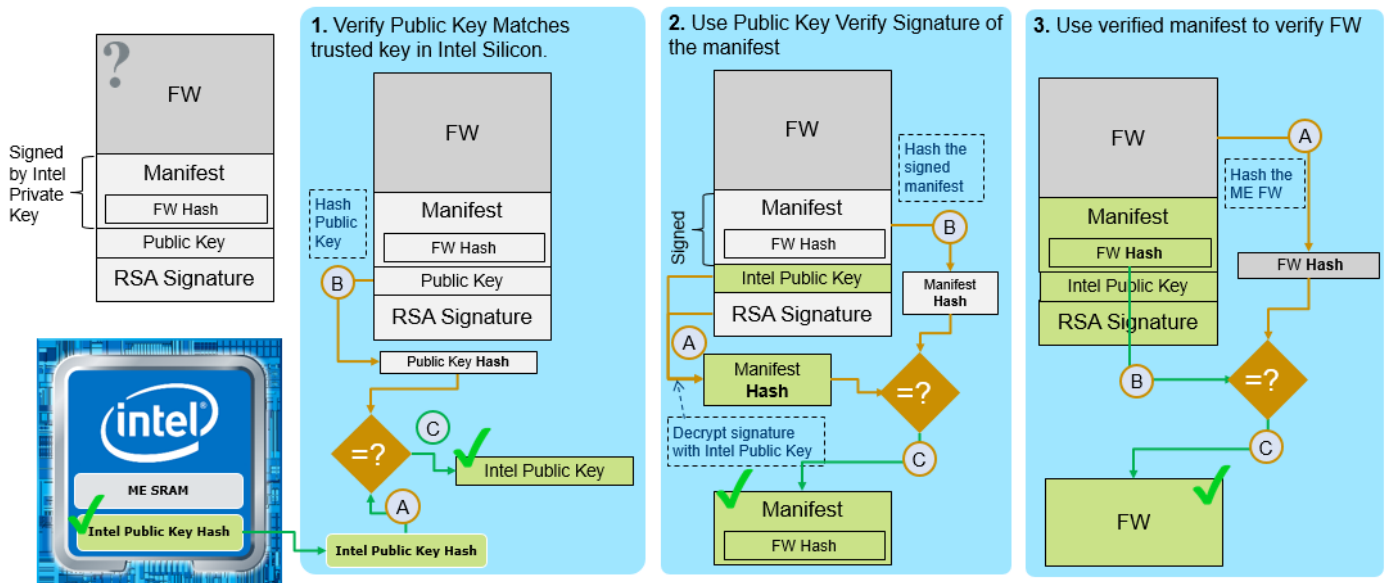
- Public key in RBE and ROT KM manifest header will be hashed and compared with the public key hash in ROM.
- Public key in OEM KM manifest will be hashed and compared with public key hash in OEM PPF.
- Public key in OEM ISH will be hashed and compared with public key hash for ISH in OEM KM when present there. If not present there, Intel ISH public key in manifest will be hashed and compared with public key hash for ISH in ROT KM.

2. **Use Public Key to Verify Signature**

Once public key in manifest was verified, it is used to decrypt the signature. This will produce a hash of the manifest section without the public key and signature. The manifest in the binary is hashed and compared with the decrypted signature output. If these hashes of the manifest equal, then the manifest has been authenticated.

3. **Use Verified Manifest to Verify FW**

Manifest has been verified, therefore anything within it can be trusted, including the hash of the original FW binary. The original FW is hashed and compared with the hash of the FW in the manifest to authenticate the FW. If the hashes equal, the component is fully authenticated and can be loaded or used to authenticate the next step in the chain.



4 *What Can Be OEM Signed*

The OEM signing infrastructure is available to support authenticating OEM signed FW. The main use cases are:

- ISH
- Audio
- Camera (only resigning of Intel IP)
- Descriptor
- FITC.cfg
- Token

There are some differences between the signing flows for each of the items listed above, therefore, please refer to the signing instructions below for guidance on the necessary steps taken for signing each one.

An OEM Key Manifest must be created and signed to hold the keys of any or all the items listed above.

5 How to Sign

5.1 High Level Signing of OEM Components

1. Generate PKI key pairs and the public key hash for:
 - a. Each component to be signed by OEM
 - b. The OEM Key Manifest(When production signing, keys used to generate signature should be from secure server. See production signing section.)
2. Use the Intel® MEU tool to add to each binary a manifest, signature, and where relevant also add metadata or compress the binary. (When production signing, keys used to generate signature should be from secure server.)
3. Create an OEM Key Manifest¹, including within it the public key hash of each of the created keys for the correct corresponding component, and use the Intel MEU to manifest/sign it.

Note: The order in which steps 2 and 3 are executed does not matter.
4. Enter the desired image components to the MFIT tool. This should include the Intel components of the image as well as any OEM signed component, the OEM KM and the public key hash corresponding to the private key used to sign the OEM KM.

At EOM (End of Manufacturing)/closemnf process, the public key hash value will be burned into the HW FPFs permanently.
5. For debug use-cases, you may add an OEM debug token to Intel FIT.

5.2 Quick List of Signing Commands

1. Generate a local private/public key pair

The Intel tools are designed to work together with the open source OpenSSL tool (version 1.0.2b or later), which generates key pairs in the RSA-3072 PKCS-1.5 format. **This is the only key format which is supported for the Intel IFWI image signing flow!** Although other tools which generate key pairs in this format can be used for signing, Intel tools currently do not interface with any other tool, and if you choose to use a different tool, Intel cannot provide support.

The OpenSSL tool is not provided by Intel, it must be installed separately. One source for the OpenSSL binaries is [Shining Light Productions](#), the "Light" version is sufficient. Ensure that OpenSSL.exe can be run in the directory in which it is installed,

¹ OEM KM is optional. OEMs who do not wish to use OEM KM may keep OEM Public Key hash as zeros in FIT tool.
If flashing an image without OEM KM at the time of EOM, the platform will never be able to contain an OEM KM.

How to Sign

and it is able to create output files there as well, otherwise you may see errors when executing some of the commands.

You can generate a private key by running the following command from the CLI:

- a. Generate privateKey.pem:
`Openssl.exe genrsa -out <privateKey.pem> 3072`
- b. Generate publicKey.pem:
`Openssl.exe rsa -in <privateKey.pem> -pubout -out <publicKey.pem>`

Note: Generate a key pair for each component to be signed, as well as for OEMKeyManifest. Or sign all with the same key pair.

2. Generate meu_config.xml

`meu.exe -gen meu_config`

- a. Update path to openssl.exe
- b. Update path to privateKey.pem
- c. Update path to LZMA (If signing ISH)
 (LZMA tool can be downloaded from [here](#))

3. Generate PubKeyHash.bin

`meu.exe -keyhash <pubKeyHash> -key <privateKey.pem>`

Note: There are additional commands listed in the next chapter for creating the public key hash manually with Openssl, or using MEU to extract it from a binary or along with the signing command.

4. Generate the necessary xml for component being signed:

`meu.exe -gen [codepartition] [codepartitionmeta]
 [oemunlocktoken] [dnximagerecovery]`

(Need a separate code partition file for each IUP or capability.)

Update the value field under:

- (1) Name
- (2) Usage (taken from value_list)
- (3) Version (see versioning section below)
- (4) InputFile (raw bin)

5. Generate OEMKeyManifest.xml

`meu.exe -gen OEMKeyManifest`

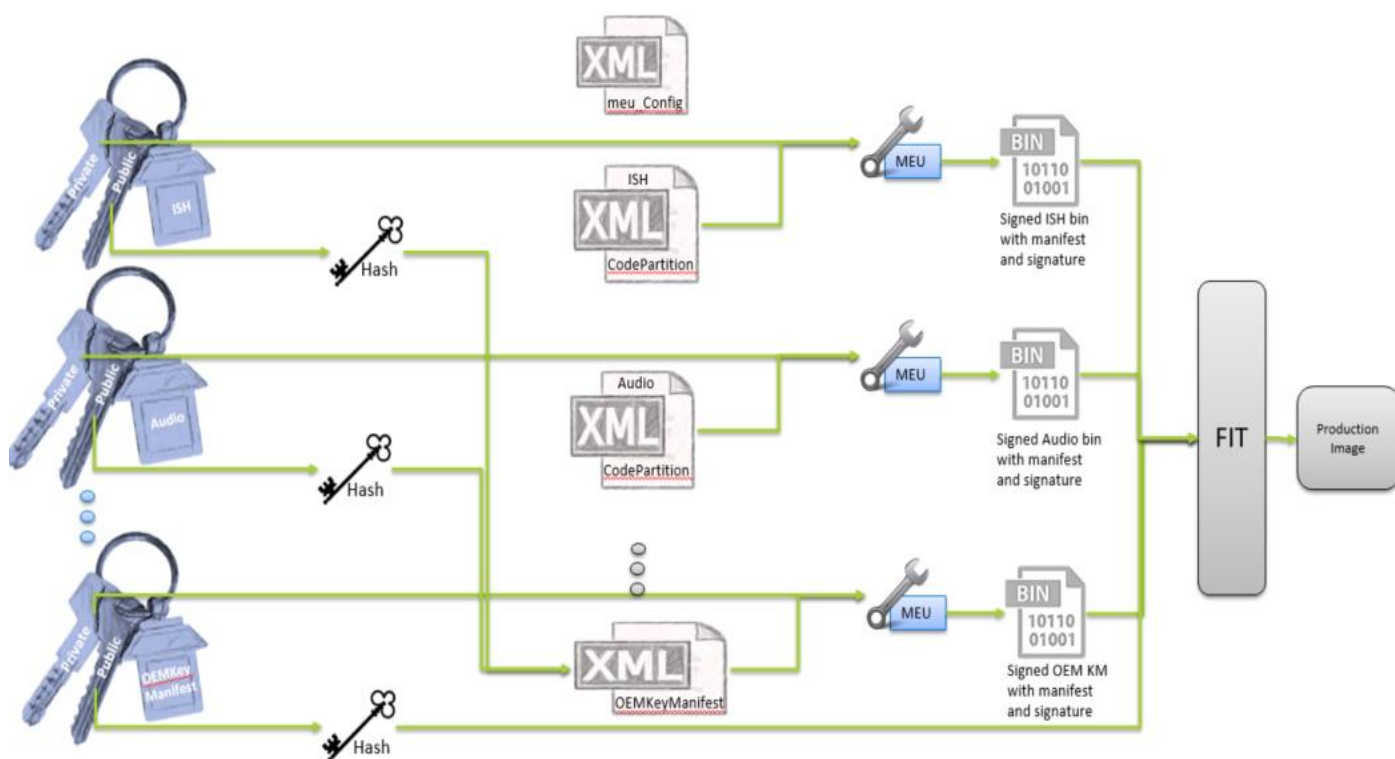
Update value field under:

- (1) KeyManifestId
- (2) (If necessary) SecurityVersionNumber
- (3) Usage
- (4) HashBinary

6. Generate CodePartition_signed.bin (Signs the Codepartition.xml)

```
meu.exe -f CodePartition.xml -o
<CodePartition_signed.bin> -key <privateKey.pem>
```

- 7. Generate OEMKeyManifest_signed.bin** (Signs the OEMKeyManifest.xml)
meu.exe -f OEMKeyManifest.xml -o
<OEMKeyManifest_signed.bin> -key <privateKey.pem>



5.3 Extended Signing Commands, Detailed Instructions and MEU Abilities

5.3.1 Additional ways to generate public key hash

Using MEU:

1. Extract public key hash from signed binary:

```
meu.exe -keyhash <output hashfile> -f <input.bin>
```

Example:

```
# meu.exe -keyhash temp/hash -f iunp.bin
=====
Intel(R) Manifest Extension Utility. Version: 16.0.0.xxxx
Copyright (c) 2013 - 2021, Intel Corporation. All rights
reserved.
MM/DD/YYYY - HH:MM:SS am
=====
Command Line: meu -keyhash temp/hash -f iunp.bin
Log file written to meu.log
Loading XML file: C:/Users/meu_config.xml
Public Key Hash Value:
  14 05 A8 A4 EB 1C 8A C2 51 19 7D 85 96 14 09 FF 15 FD CD
23 D3 25 CC DD 88 D2 17 5C DE 3B 27 36
Public Key Hash Saved to:
  temp\hash.bin
  temp\hash.txt
Program terminated.
-----
```

2. Generate public key hash along with the signing command:

```
meu.exe -keyhash <output hashfile> -f <input.xml> -o
<output.bin>
```

Manually with Openssl:

1. Extraction from the public or private key:

- 1.1. If using the public key:

```
openssl.exe rsa -in public.pem -text -noout -pubin
```

- 1.2. If using the private key:

```
openssl.exe rsa -in private.pem -text -noout
```

- a. Copy the modulus (excluding any leading bytes that are all 0s)
- b. Reverse the modulus byte order (Use excel to paste all the bytes on different rows into a column, then put ascending numbers in another column and do a reverse sort on the numbers)
- c. Paste the reverse byte modulus into a new file <new file> in a hex editor
- d. Copy the exponent following the modulus into the new file (make sure it is little endian)

Hash the new file using

```
openssl.exe dgst -sha384 <new file>
```

2. Extraction from a manifest signed with the keys, by MEU
 - a. Open a signed file that MEU has created in a hex editor
 - b. Search for the string "\$MN2", then move 100 bytes after the start of "\$MN2" (this will be the start of the modulus + exponent)
 - c. Extract the following 260 bytes to a new file <new file>
 - d. Hash the new file using openssl:


```
openssl.exe dgst -sha384 <new file>
```

The public key hash is a readable string, and can be copied and pasted from the text file as needed.

5.3.2 Versioning of Signed Components

5.3.2.1 Major, Minor, Hotfix, Build

All XMLs generated by MEU contain a field for setting the version in the manifest of the binary to be signed.

```
<SecurityVersionNumber value="0x00000000" help_text="The security version number of the OEM Key Manifest" />
<VersionMajor value="0x0000" help_text="Indicates the major number in the version numbering" />
<VersionMinor value="0x0000" help_text="Indicates the minor number in the version numbering" />
<VersionHotfix value="0x0000" help_text="Indicates the hotfix number in the version numbering" />
<VersionBuild value="0x0000" help_text="Indicates the build number in the version numbering" />
```

OEMs are required to define these versions so the component can be identified by its version. Versions are updated based on the changes made, with the following rule of thumb in mind:

Major	A major change in the component or design
Minor	A minor change to the component
Hotfix	If the new component is basically the same as before, but includes a hotfix
Build	Incremented any time the component is rebuilt again for whatever reason

Here is the breakdown of the versioning as an examples taken from CSME:

VersionMajor: 16 (when CSME version **16**.0.0.1000)

VersionMinor: 8 (when CSME version 16.**0**.0.1000)

VersionHotfix: 50 (when CSME version 16.0.0.1000)

VersionBuild: 3399 (when CSME version 16.0.0.**1000**)

5.3.2.2 Security Version Number (SVN)

The security version number (SVN) starts at 1 for production IPs. It is used as a security measure to block the loading of versions with security vulnerabilities. On a platform which contains an IP with SVN = x, upgrade is allowed to versions with SVN=x or SVN>x.

Therefore:

- To allow downgrade to the previous IP versions, keep SVN the same value as the previous version.
- To block downgrade to the previous IP versions, increase the SVN.

For example, in machine that has a component with version 1.1.0.2 and SVN 2, the following applies:

Version	SVN Value	Can it be updated?
1.0.0.1	1	No, the SVN value is lower
1.1.0.1	2	Yes, same SVN value
1.2.0.0	3	Yes, higher SVN value

5.3.2.3 Hardware Anti-Rollback (ARB)

The SVN value of OEM KM can be stored in fuses to provide a hardware level protection of anti-rollback. HW ARB requires OEMs to invoke a HECI command to set the SVN value into fuses.

For information on how to commit the SVN value to FPF, please refer to the BIOS writers guide.

This feature can extend to securing the rollback of any IP authenticated by the OEM KM by following these steps:

1. Raise the SVN value of any IP
2. Use a new production key to sign the IP with the raised SVN value
3. Enter the corresponding new public key hash into the OEM KM for the relevant IP
4. Raise the SVN value in the OEM KM
5. Stitch the updated components into a new image
6. Use full FW Update to apply the new image
7. Apply the ARB SVN to the fuses by invoking the relevant HECI command.

5.4 Intel® Manifest Extension Utility (Intel® MEU)

The Intel® Manifest Extension Utility (MEU) receives as input a firmware binary created by a 3rd party and outputs an independent-updateable partition (IUP) that is signed.

The Intel® Manifest Extension Utility (MEU) requires administrator privileges to run under Windows* OS.

The Intel® MEU tool completes the following steps:

- Creates an Independent Updatable Partition (IUP) by adding manifest and meta-data information to the firmware.
- Calls an external LZMA tool for compression of the ISH binary
- Calls the signing infrastructure tool to sign the partition.

5.4.1 Usage

The executable can be invoked by:

```
meu.exe [-exp] [-h|?] [-3rdparty] [-version|ver] [-binlist]
[-o] [-f] [-gen] [-cfg] [-decomp] [-save] [-w] [-s] [-d] [-u1] [-u2]
[-u3] [-mnver] [-mnpv] [-mndebug] [-st] [-stp] [-key] [-noverify]
[-keyhash] [-resign] [-export] [-import] [-printman]
```

Option	Description
-H or -?:	Displays the list of command line options supported by the Intel® MEU tool.
-3rdparty	Displays 3rd party software credits.
-EXP	Shows examples about how to use the tools.
-ver version	Shows the version of the tools.
-binlist	Displays a list of supported binary types.
-o <filename>	Overrides the output file path.
-f <filename>	Specifies input XML file.
-gen <type>	Specifies the binary type for which to generate a template XML file.
-cfg <filename>	Overrides the path to the tool config XML file.
-decomp <type>	Specifies the binary type to use for decomposition.
-save <filename>	Specifies the output XML path.
-w <path>	Overrides the \$WorkingDir environment variable.
-s <path>	Overrides the \$SourceDir environment variable.

Option	Description
-d <path>	Overrides the \$DestDir environment variable.
-u1 <path>	Overrides the \$UserVar1 environment variable.
-u2 <path>	Overrides the \$UserVar2 environment variable.
-u3 <path>	Overrides the \$UserVar3 environment variable.
-mnver <value>	Overrides the version of the output binary. (Format: Major.Minor.Hotfix.Build)
-mnpv <true false>	Overrides the PV flag in the output binary's manifest(s).
-mndebug <true false>	Overrides the debug flag in the output binary's manifest(s).
-key <path>	Overrides the signing key in the tool config XML file.
-st <tool>	Overrides SigningTool in the tool config XML file.
-stp <path>	Overrides SigningToolPath in the tool config XML file.
-noverify	Skips verification of generated manifest signature.
-keyhash <path>	Exports the public key hash to a file.
-resign <indices 'all'>	Resigns manifest(s) in a binary.
-export <indices 'all'>	Exports manifest(s) from a binary.
-import <path>	Imports manifest(s) into a binary.
-printman <indices 'all'>	Prints manifest(s) information from a binary

5.4.2 Examples

5.4.2.1 Generate Configuration XML Template

To get started using Intel MEU for signing, it is mandatory to set some configurations for the tool. To do this, run the following command:

```
# meu -gen meu_config
```

This will generate a default configuration xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<CodePartition version="2.5" >
  <Name value="ISHC" help_text="Name to use in the output binary's directory. Maximum length is 4 characters." />
  <Length value="0x0" help_text="Length of output binary, extra space will be filled with 0xFF's. If length is smaller than required, an
error will be reported. If set to 0, the length will be computed as needed by the tool." />
  <Usage value="IsbManifest" value_list="CseBugManifest,,CseMainManifest,,PchManifest,,UsbTypeCIOMManifest,,UsbTypeCIphyManifest,,
UsbTypeCTMManifest,,WoodManifest,,LoolManifest,,IntelUtokManifest,,SPHYManifest,,PohManifest,,SamManifest,,PphyManifest,,GbstManifest,,
BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAveImageManifest,,cAveImageManifest,,IfwManifest,,OsBootLoaderManifest,,
OsKernelManifest,,OemSmipManifest,,IsbManifest,,OemDebugManifest,,OemKeyManifest,,SilentLakeVmmManifest,,OemDnxIfwManifest"
help_text="Indicates the type of data contained in this binary. This value is used during signature verification to validate the public key." />
  <VendorId value="0x0000" />
  <InstanceId value="0x1" />
  <PartitionFlags value="0x00000000" help_text="Refers to flags relevant to manifest for a specific partition. Bit 0 should be set on for
partition of PV or post PV release." />
  <PartitionVersion value="0x10000000" />
  <VersionControlNumber value="0x00000000" />
  <SecurityVersionNumber value="0x00000000" />
  <VersionMajor value="0x0" label="Version Major" help_text="Used to manually set the Major Version field in the manifest" />
  <VersionMinor value="0x0" label="Version Minor" help_text="Used to manually set the Minor Version field in the manifest" />
  <VersionHotfix value="0x0" label="Version Hotfix" help_text="Used to manually set the Hotfix Version field in the manifest" />
  <VersionBuild value="0x0" label="Version Build" help_text="Used to manually set the Build Version field in the manifest" />
  <VersionExtraction>
    <Enabled value="false" value_list="true,,false" help_text="If enabled, the version details will be extracted from the InputFile binary at
the offsets specified. If disabled, the version must be specified manually." />
    <InputFile value="" help_text="Binary file from which to extract the version details." />
    <VersionMajorByteOffset value="0" help_text="Offset of Major Version number's LSB in InputFile." />
    <VersionMinorByteOffset value="0" help_text="Offset of Minor Version number's LSB in InputFile." />
    <VersionMinorByteOffset value="0" help_text="Offset of Minor Version number's LSB in InputFile." />
    <VersionHotfixByteOffset value="0" help_text="Offset of Hotfix Version number's LSB in InputFile." />
    <VersionHotfixByteOffset value="0" help_text="Offset of Hotfix Version number's LSB in InputFile." />
    <VersionBuildByteOffset value="0" help_text="Offset of Build Version number's LSB in InputFile." />
    <VersionBuildByteOffset value="0" help_text="Offset of Build Version number's LSB in InputFile." />
  </VersionExtraction>
  <CPModules>
    <CPDataModule name="ish_main">
      <InputFile value="ish_main.bin" help_text="Path to binary file to load for this module's data." />
      <CompressionType value="LZMA" value_list="NOT_COMPRESSED,,LZMA" help_text="Select compression type for this module." />
      <ProcessId value="0xf6" />
    </CPDataModule>
  </CPModules>
</CodePartition>
```

The XML is generated with a default of using Openssl as the signing tool. The user must enter the correct path, to the signing tool executable, under the value of SigningToolPath:

To generate the manifest structure without the signature and public key, set the signing tool value to 'Disabled':

```
<SigningTool value="Disabled"
Value_list="Disabled,,OpenSSL" label="Signing Tool"
help_text="Select tool to be used for signing, or disable
signing." />
```

When signing is set to Disabled, there is no need to add the Openssl path as indicated above.

If using a single private key to sign several components (such as for R&D purposes), the private key path may be entered into this XML instead of the signing command:

```
<PrivateKeyPath value="$WorkingDir\private.pem"
label="Private Key Path" help_text="Path to private RSA key
(in PEM format) to be used for signing. Key is required if
using OpenSSL." />
```

If the PrivateKeyPath value is left blank here, the private key will be mandatory in the signing command.

When signing ISH component, it is mandatory to compress the binary using LZMA tool. This is done by setting the LZMA tool path in the configuration XML:

```
<LzmaToolPath value="" label="LZMA Tool Path"
help_text="Path to lzma tool executable." />
```

For signing any other component, leave this value empty as default.

5.4.2.2 Generate Code partition XML

Code partition XML is used to set the manifest data for ISH. Generate a code partition XML to manifest, compress and sign ISH with the following command:

```
# meu -gen CodePartition
```

This will generate a default codepartition.xml file:

ISH is the only OEM signed component using the code partition XML, therefore the default values are set for ISH.

It is important to configure the versioning and add the correct path to the ISH binary file in:

```
<InputFile value="ish_main.bin" help_text="Path to binary file to load for this module's data." />
```

And leave the compression setting value to LZMA, so the binary is compressed. (The LZMA path must be entered in the meu_config.xml.)

Once the codepartition.xml has been edited to include all the required input fields, MEU can be run with the xml as input to manifest and sign it with the private key created for this purpose.

5.4.2.3 Generating Code Partition Meta

The IUnit (camera) and aDSP (Audio) FW binaries use the codepartitionmeta.xml file to manifest and sign their binaries. Meta in the file name refers to the metadata added for these components.

Generate the code partition meta xml file with the following command:

```
# meu -gen CodePartitionMeta
```

This will generate a default codepartitionmeta.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<CodePartitionMeta version="2.5" >
  <Name value="IUNP" help_text="Name to use in the output binary's directory. Maximum length is 4 characters." />
  <Length value="0x0" help_text="Length of output binary, extra space will be filled with 0xFF's. If length is smaller than
required, an error will be reported. If set to 0, the length will be computed as needed by the tool." />
  <Usage value="" value_list="CseBupManifest,,CseMainManifest,,PmcManifest,,UsbTypeCIOMManifest,,UsbTypeCNphyManifest,,
UsbTypeCTBTManifest,,WoodManifest,,LoclManifest,,IntelUtokManifest,,SPHYManifest,,PchManifest,,SamfManifest,,PphyManifest,,
GbstManifest,,BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0Manifest,,cAvsImage1Manifest,,
IfwiManifest,,OsBootLoaderManifest,,OsKernelManifest,,OemSmipManifest,,IshManifest,,OemDebugManifest,,OemKeyManifest,,
SilentLakeVmmManifest,,OemDnxIfwiManifest" help_text="Indicates the type of data contained in this binary.
This value is used during signature verification to validate the public key." />
  <VendorId value="0x0000" help_text="32-bit Vendor ID value. (ex. Intel=0x8086)" />
  <InstanceId value="0x1" />
  <PartitionFlags value="0x00000000" help_text="Refers to flags relevant to manifest for a specific partition.
Bit 0 should be set on for partition of PV or post PV release." />
  <PartitionVersion value="0x10000000" />
  <VersionControlNumber value="0x00000000" />
  <SecurityVersionNumber value="0x00000000" />
  <VersionMajor value="0x0" label="Version Major" help_text="Used to manually set the Major Version field in the manifest" />
  <VersionMinor value="0x0" label="Version Minor" help_text="Used to manually set the Minor Version field in the manifest" />
  <VersionHotfix value="0x0" label="Version Hotfix" help_text="Used to manually set the Hotfix Version field in the manifest" />
  <VersionBuild value="0x0" label="Version Build" help_text="Used to manually set the Build Version field in the manifest" />
  <VersionExtraction>
    <Enabled value="false" value_list="true,false" help_text="If enabled, the version details will be extracted from the
InputFile binary at the offsets specified. If disabled, the version must be specified manually." />
    <InputFile value="" help_text="Binary file from which to extract the version details." />
    <VersionMajorByteOffset value="0" help_text="Offset of Major Version number's LSB in InputFile." />
    <VersionMajorByteOffset value="0" help_text="Offset of Major Version number's MSB in InputFile." />
    <VersionMinorByteOffset value="0" help_text="Offset of Minor Version number's LSB in InputFile." />
    <VersionMinorByteOffset value="0" help_text="Offset of Minor Version number's MSB in InputFile." />
    <VersionHotfixByteOffset value="0" help_text="Offset of Hotfix Version number's LSB in InputFile." />
    <VersionHotfixByteOffset value="0" help_text="Offset of Hotfix Version number's MSB in InputFile." />
    <VersionBuildByteOffset value="0" help_text="Offset of Build Version number's LSB in InputFile." />
    <VersionBuildByteOffset value="0" help_text="Offset of Build Version number's MSB in InputFile." />
  </VersionExtraction>
  <CodePartitionMetadata>
    <Name value="iunit.met" help_text="Name to use as metadata filename in output binary. Maximum length is 12 characters." />
    <InputFile value="iunit_met.bin" help_text="Local path to metadata binary file" />
  </CodePartitionMetadata>
  <CPMModules>
    <CPMDataModule name="iunit">
      <InputFile value="iunit.bin" help_text="Path to binary file to load for this module's data." />
      <CompressionType value="NOT_COMPRESSED" value_list="NOT_COMPRESSED,,LZMA" help_text="Select compression type
for this module." />
    </CPMDataModule>
  </CPMModules>
</CodePartitionMeta>
```

The default codepartitionmeta.xml file is set to IUnit (camera) component but can be edited for cAVS (Audio) as well.

To sign IUnit, set the usage value to iUnitMainFwManifest from the value_list, set versioning and enter the path to the main IUnit binary and metadata binary file:

```
<CodePartitionMetadata>
  <Name value="iunit.met" help_text="Name to use as metadata filename in output binary. Maximum length is 12
characters." />
  <InputFile value="iunit_met.bin" help_text="Local path to metadata binary file" />
</CodePartitionMetadata>
<CPMModules>
  <CPMDataModule name="iunit">
    <InputFile value="iunit.bin" help_text="Path to binary file to load for this module's data." />
    <CompressionType value="NOT_COMPRESSED" value_list="NOT_COMPRESSED,,LZMA" help_text="Select compression type
for this module." />
  </CPMDataModule>
</CPMModules>
```

Note: Loading of an OEM IUnit is not supported. Signing of IUnit is only relevant for re-signing Intel IUnit component.

To edit the file for Audio signing, change the Name value to "CAVS", set the usage value to cAvsImage0Manifest from the value_list,

```
<CodePartitionMetadata>
  <Name value="cavs.met" help_text="Name to use as metadata filename in output binary. Maximum length is 12 characters." />
  <InputFile value="cavs_met.bin" help_text="Local path to metadata binary file" />
</CodePartitionMetadata>
<CPMModules>
  <CPMDataModule name="cavs">
    <InputFile value="cavs.bin" help_text="Path to binary file to load for this module's data." />
    <CompressionType value="NOT_COMPRESSED" value_list="NOT_COMPRESSED,,LZMA" help_text="Select compression type for this module." />
  </CPMDataModule>
```

configure the versioning and set the name value and input value of binaries to correspond to the Audio component:

Once the codepartitionmeta.xml has been edited to include all the required input fields, MEU can be run with the xml as input to manifest and sign it with the private key created for this purpose.

5.4.2.4 Secure Tokens (OEM Unlock Tokens)

The OEMUnlockToken binary is authenticated by the Intel CSME FW. OEMs who wish to use this feature need to create token, sign it with OEM private key and include the public key hash in the OEM KM for OemUnlockToken. To create such token, the OEM needs to generate xml for it using the following command:

```
# meu -gen OemUnlockToken
```

This will generate a default oemunlocktoken.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<OemUnlockToken version="2.5" >
  <ExpirationSeconds value="0x00278D00" help_text="Time from Part ID generation to Token expiration (in seconds)." />
  <PartIdsPath value="" help_text="Path to directory containing Part ID binaries." />
  <TokenFlags>
    <PartRestricted value="Yes" value_list="Yes,,No" />
    <AntiReplayProtected value="Yes" value_list="Yes,,No" />
    <TimeLimited value="Yes" value_list="Yes,,No" />
    <SingleBoot value="No" value_list="No,,Yes" />
  </TokenFlags>
  <TokenKnobs>
    <OemUnlockKnob value="DoNothing" value_list="DoNothing,,OemUnlockEnabled" />
    <IshGdbDebugKnob value="DoNothing" value_list="DoNothing,,IshGdbSupportEnabled" />
    <BootGuardKnob value="DoNothing" value_list="DoNothing,,BootGuardDisabled,,BootGuardNoEnforcement,,BootGuardNoTimeouts,,BootGuardNoEnforcementAndTimeouts" />
    <OemBiosPayload value="0x00000000" />
    <DnXCapabilitiesKnob>
      <CapabilitiesBitmap value="" value_list="EnableGetNvmProperties,,EnableNvmConfiguration,,EnableClearPlatformConfiguration,,EnableWritingNvmContent,,EnableReadingNvmContent"
        help_text="Bit field to enable/disable DnX knob capabilities." />
    </DnXCapabilitiesKnob>
    <EnableCseTraceKnob value="DoNothing" value_list="DoNothing,,CseTraceDisabled,,CseTraceEnabled" />
    <CancelOemSigningKnob value="DoNothing" value_list="DoNothing,,CancelOemSigningCheck" />
  </TokenKnobs>
</OemUnlockToken>
```

There are multiple flags that can be set for the token creation:

- **PartRestricted:** Set to yes to allow token to be used on any platform where the token key hash in OEM KM authenticates that token, and token is tied to a particular platform ID.
- **Anti-Replay Protected:** Set to yes to disable a token from being re-used on the same device after new token is created. Relevant for tokens tied to a particular platform ID.
- **TimeLimited.** Set to yes to have token expire after a given time period. Anti-Replay Protected must be set for token with time expiration, because otherwise you can re-use the token after RTC clear.
- **Single Boot**

It is recommended to use to secure token with time expiration and Anti-replay flag.

In the root node you can set:

- **Expiration timeout** (if relevant)

- **Part ID path.** You can retrieve the Part ID data using Intel® FPT, by calling
FPT.exe -GETPID <file>

This will retrieve the part ID into a file. Provide the path to the directory that contains PID.bin or multiple PID binaries.

Note: Executing this command will invalidate all secure tokens with Anti-replay protection generated earlier for the given platform

In the TokenKnobs section, set the 'Knobs' for the token. These define what the token allows/disables on the platform. The knobs available vary depending on the token being created. Here is an explanation of the various knobs:

Knob	Meaning
OEM Unlock	Allow an OEM (Orange) unlock. It will enable debug interfaces to ISH and Audio
ISH GDB Debug	Enable ISH GDB support
Cancel OEM signing	CSE skips the authentication of the OEM signed FW when an OEM signed token with a knob for canceling OEM authorization is present.

Note:
BootGuardDisabled,,BootGuardNoEnforcement,,BootGuardNoTimeouts,,BootGuardNoEnforcementAndTimeouts are not supported with OEM Secure Token and should be set to DoNothing.

Once the OEMUnlockToken xml has been edited to include all the required input files the MEU can be run with the xml as input to manifest and sign it with the private key created for this purpose.

5.4.2.5 Generate OEM KM XML

The manifest file xml template can be generated using the following command:

```
# meu -gen OEMKeyManifest
```

This will generate a default oemunlocktoken.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<OEMKeyManifest version="2.5" >
  <Name value="OEMP" help_text="Name to use in the output binary's directory. Maximum length is 4 characters. " />
  <Length value="0x0" help_text="Length of output binary, extra space will be filled with 0xFF's. If length is smaller than
required, an error will be reported. If set to 0, the length will be computed as needed by the tool." />
  <OemId value="0x0000" help_text="ID of the OEM creating the Key Manifest" />
  <KeyManifestId value="0x1" help_text="ID number of the Key Manifest. This is matched by the verifier against the value stored
in the platform's FPF." />
  <InstanceId value="0x1" help_text="Refers to the instance of the Key Manifest at hand" />
  <PartitionFlags value="0x00000000" help_text="Refers to flags relevant to manifest for a specific partition." />
  <PartitionVersion value="0x10000000" help_text="Refers to the version of the partition of the relevant manifest" />
  <VendorId value="0x8086" help_text="Shows the vendor ID owning the Key Manifest at hand" />
  <VersionControlNumber value="0" help_text="The VCN is incremented whenever a change is made to the FW making it incompatible
from an update perspective with previous FW releases" />
  <SecurityVersionNumber value="0x00000000" help_text="The security version number of the OEM Key Manifest" />
  <VersionMajor value="0x0000" help_text="Indicates the major number in the version numbering" />
  <VersionMinor value="0x0000" help_text="Indicates the minor number in the version numbering" />
  <VersionHotfix value="0x0000" help_text="Indicates the hotfix number in the version numbering" />
  <VersionBuild value="0x0000" help_text="Indicates the build number in the version numbering" />
  <KeyManifestEntries>
    <KeyManifestEntry>
      <Usage value="BootPolicyManifest | IfwiManifest" value_list="BootPolicyManifest, iUnitBootLoaderManifest,
iUnitMainFwManifest, ,CavsImageManifest, ,IfwiManifest, ,OsBootLoaderManifest, ,OsKernelManifest, ,OemSmipManifest,
,IshManifest, ,OemDebugManifest, ,SilentLakeVmmManifest, ,OemAttestationManifest, ,OemDnxIfwiManifest,
,OemDescriptorManifest" />
      <HashBinary value="pubkey_hash.bin" help_text="Path to binary file containing Public Key Hash" />
    </KeyManifestEntry>
  </KeyManifestEntries>
</OEMKeyManifest>
```

Edit KeyManifestId field to a value other than zero. This value will be entered into MFIT and burned to an FPF.

Important!

The KeyManifestId field must be given a non-zero value. It is critical that the matching field in MFIT is also changed to the exact same non-zero value. This field will be burned into an FPF and used to validate the OEM Key Manifest on platform boot.

When updating an image with a new image, the new OEM KM must have the same non-zero value as well.

Extra 'KeyManifestEntry' nodes should be added for each file that has a unique key hash to be entered. If several files share the same key, they can be included within the same node, as in the default xml template.

So, for example, if the OEM Key Manifest will have several IPs signed with the same key, eg:

- IshManifest, iUnitBootLoaderManifest & iUnitMainFwManifest with key 1

It would appear as follows:

```
<KeyManifestEntries>
  <KeyManifestEntry>
    <Usage value="IshManifest | iUnitBootLoaderManifest | iUnitMainFwManifest" value_list=
      "BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0Manifest,,cAvsImage1Manifest,,IfwiManifest,,OsBootL
oaderManifest,,OsKernelManifest,,OemSmipManifest,,IshManifest,,OemDebugManifest,,SilentLakeVmmManifest,,OemDnxIfwiManifest" />
    <HashBinary value="pubkey_hash.bin" help_text="Path to binary file containing Public Key Hash (Must be 32 bytes)" />
  </KeyManifestEntry>
</KeyManifestEntries>
```

If the OEM Key Manifest has a separate key for each IP, eg:

- IshManifest with key 1
- iUnitBootLoaderManifest & iUnitMainFwManifest with key 2

It would appear as follows:

```
<KeyManifestEntries>
  <KeyManifestEntry>
    <Usage value="IshManifest" value_list=
      "BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0Manifest,,cAvsImage1Manifest,,IfwiManifest,,OsBoot
LoaderManifest,,OsKernelManifest,,OemSmipManifest,,IshManifest,,OemDebugManifest,,SilentLakeVmmManifest,,OemDnxIfwiManifest" />
    <HashBinary value="pubkey_hash1.bin" help_text="Path to binary file containing Public Key Hash (Must be 32 bytes)" />
  </KeyManifestEntry>
  <KeyManifestEntry>
    <Usage value="iUnitBootLoaderManifest | iUnitMainFwManifest" value_list=
      "BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0Manifest,,cAvsImage1Manifest,,IfwiManifest,,OsBoot
LoaderManifest,,OsKernelManifest,,OemSmipManifest,,IshManifest,,OemDebugManifest,,SilentLakeVmmManifest,,OemDnxIfwiManifest" />
    <HashBinary value="pubkey_hash2.bin" help_text="Path to binary file containing Public Key Hash (Must be 32 bytes)" />
  </KeyManifestEntry>
</KeyManifestEntries>
```

Once the OEM Key Manifest xml has been edited to include all the required entries and hashes, the MEU can be run with the xml as input to manifest and sign it with the private key created for this purpose.

5.4.2.6 Empty OEM KM

Once a platform is manufactured without an OEM KM, the image can never be updated with an OEM KM. Therefore, if an OEM does not have use for an OEM KM at the time of manufacturing, Intel still advises that an OEM KM be added to the image. This can be done by adding an empty OEM KM (no entries), which works as a placeholder in the image. At a later time, once platform is in field, the image can be updated via FWUpdate, to one with an OEM KM with relevant keys. (eg. OEM debug token keys, adding an OEM audio driver etc.)

To do so, leave empty quotes for Usage Value, and provide a bin file with zeros for the HashBinary value.

Eg:

```
<KeyManifestEntries>
  <KeyManifestEntry>
    <Usage value="" value_list=
      "BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0Manifest,,IfwiManifest,,OsBootLoaderManifest,,OsKernelManifest,,OemSmipM
anifest,,IshManifest,,OemDebugManifest,,SilentLakeVmmManifest,,OemDnxIfwiManifest" />
    <HashBinary value="HashBinary0.bin" help_text="Path to binary file
containing Public Key Hash)" />
  </KeyManifestEntry>
</KeyManifestEntries>
```


5.4.2.7 Signing Command with Input XML

Once the desired XML has been edited to include all the required entries, this command will create the manifested and signed partition using MEU.

```
# MEU.exe -f <XML_FILE.xml> -o <Output_file_Name.bin>
```

If a private key was not specified in the MEU_config.xml, or if a different key is to be used, add the key to the signing command as follows:

```
# MEU.exe -f <XML_FILE.xml> -o <Output_file_Name.bin> -key <privateKey>
```

5.4.2.8 Intel® MEU Binlist

Intel MEU supports manifesting and signing several different file types, as listed above. To see the full list, run the following:

```
# meu.exe -binlist
```

```
Intel(R) Manifest Extension Utility. Version: 16.0.0.1287
Copyright (c) 2013 - 2021, Intel Corporation. All rights reserved.
1/31/2021 - 8:47:15 pm
```

```
Command Line: \Desktop\meu.exe -binlist
```

The following binary types can be generated by this tool. A template XML file can be generated for a given type using the -gen switch.

Type	Description
meu_config	- Template tool config file (meu_config.xml)
CodePartition	- Generic Updateable Code Partition
CodePartitionMeta	- Updateable Code Partition with user-provided Metadata
DnxRecoveryImage	- DNX Recovery IFWI Image for large images
OEMKeyManifest	- OEM Key Manifest
OemUnlockToken	- OEM Unlock Token
TcssPartition	- Updateable TCSS Partition with auto Metadata

```
Program terminated.
```

5.4.2.9 Intel® MEU Decomposition

Intel MEU is able to decompose a manifested and signed binary returning it to the original state it was in before the Intel MEU added a manifest and/or signature. This provides an xml detailing the decomposition. This xml can later be used again as input to the Intel® MEU to recreate the signed binary. The -decomp command also requires the binary type as its first parameter.

To decompose an OEM Key Manifest binary:

```
# meu -decomp OEMKeyManifest -f <input.bin> -save <decomp_KM.xml>
```

To decompose a codepartition Manifest binary:

```
# meu -decomp codepartition -f <input.bin> -save
<decomp_partition.xml>
```

To decompose a codepartitionmeta Manifest binary:

```
# meu -decomp codepartitionmeta -f <input.bin> -save
<decomp_meta.xml>
```

To decompose an oemunlocktoken Manifest binary:

```
# meu -decomp oemunlocktoken -f <input.bin> -save
<decomp_token.xml>
```

5.4.2.10 Intel® MEU Re-sign

Intel® MEU is able to re-sign a binary that has already been signed. This is very useful when changing the signing keys – the relevant binary files just need to be re-signed.

```
# meu.exe -resign -f <input.bin> -o <output.bin> [-key
<privatekey.pem>]
```

Some binaries, such as full IFWI images, include multiple manifests. When calling the `-resign` option on such binaries, it is necessary to include the index of the manifest to be re-signed, or 'all' if all are to be re-signed (using the new key). If the index, or 'all' is not included, the Intel® MEU will show a full list of the manifests included in the binary:

More than one manifest was found in this file. Please provide a comma-separated list of the manifest indices you want to resign. (ex. `-resign "0,3,5"`) or specify "all" (ex. `-resign "all"`)

The following manifests were detected:

Index	Offset	Size	Name (if available)
0	0x000084058	0x000000378	RBEP.man
1	0x000094058	0x000000378	PMCP.man
2	0x0000A4580	0x000001750	FTPR.man
3	0x0000A9000	0x000000330	rot.key
4	0x0001F4000	0x000000330	oem.key
5	0x0001FB058	0x000000378	ISHC.man
6	0x00023B070	0x000000378	IUNP.man
7	0x00023D0E8	0x0000004B0	WCOD.man
8	0x0002BD0B8	0x000000448	LOCL.man
9	0x000342448	0x000000C00	NFTP.man

Error 24: Failed to resign manifest(s). Missing manifest indices list.

The Intel® MEU can then be called again including the index desired. Following the above example if the OEM KM is to be re-signed, call:

```
# meu.exe -resign 4 -f <input.bin> -o <output.bin> [-key
<privatekey.pem>]
```

6 Intel® MFIT

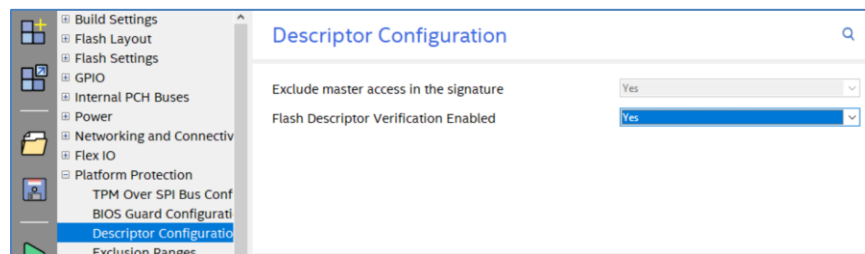
Intel® Modular Flash Image Tool is a stitching tool to combine multiple binary files into one, configuration data and add other input into a full SPI image. This document will only discuss the usage of the tool as relevant to the signing mechanism. The full image creation procedure & MFIT functionalities are detailed in the Intel® CSME Firmware Bring-Up Guide & System Tools User Guide.

6.1 Descriptor Signing

The main usage of Intel® MFIT is to stitch and generate the final image, yet since the descriptor is generated by Intel® MFIT, the tool is also used to sign it. (MEU is used behind the scenes for signing, but not visible to the user.)

The descriptor is a section of the image which contains the offset location of each region within the image. It also contains some configurations such as soft straps.

By default, the descriptor is not signed. In order to enable signing it, select yes for the Flash Descriptor Verification enabled.

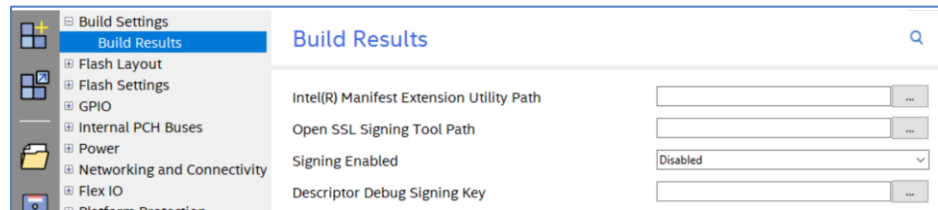


Once this is done, when the image is built, by default the manifest will be added to the descriptor in the image.

In order to production sign the manifest, MEU will need to be used to export the manifest, send it to OEM production signing server, and importing it back to the image. This flow of export and import is described later in the document.

If the user would like to debug sign the descriptor with a debug key, then user must open the build settings in MFIT, and change the "Signing Enabled" setting from "Disabled" to "Enabled", then enter the path to openssl under "Signing Tool Path".

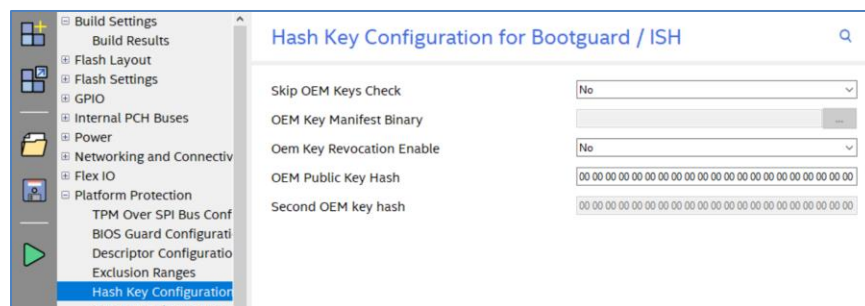
Once signing is enabled, user can add a debug signing key to the build settings:



In order to production sign the descriptor after debug signing it, follow the same instructions as above. The only change is that on the secure OEM signing server, the crypto fields will be replaced in the manifest instead of placed there for the first time.

6.2 Signing components added to Intel® MFIT

3. MFIT includes input fields allowing the input of binary files. Most are available in the Flash Layout tab.
4. Add the signed OEM KM binary into MFIT if OEM signed components are included to the image.
5. Add the Public Key Hash for OEM Key Manifest
This hash will be burned into an FPF in the FPF HW when the system closes manufacture (closemnf/EOM), and can never be changed after this stage.
Can also add a second OEM Key Hash as a security layer, to enable the first OEM public key hash to be revoked.
6. The Key Manifest ID field must be changed from 0x0 to match the value set in the OEM Key Manifest.



6.3 Intel® MFIT Manifest Version Validation

In order to prevent issues in the final image due to use of an incorrect MEU tool, MEU inserts the MEU version into the IUP manifest during the signing process. MFIT uses that data to verify that the end result image will be compatible for the image which MFIT is going to create.

The following checks are in place:

Test Title	Test Logic	Upon Failure
IUP manifest version is supported by CSE FW	IUP Manifest version (from IUP manifest) == MFIT supported manifest version	MFIT will not stitch the image. The IUP team must update MEU and resign the IUP.
MEU and MFIT are from the same project	MEU version major.minor (from IUP manifest) == MFIT version major.minor.	MFIT will not stitch the image. The IUP team must update MEU and resign the IUP.



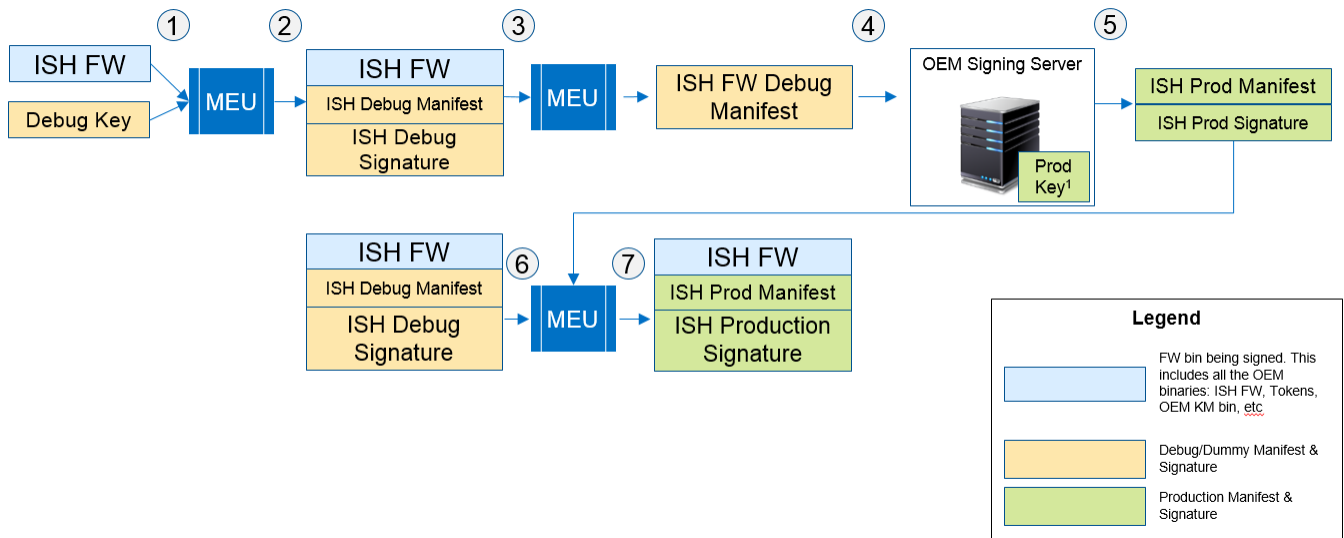
7 Production Signing

The purpose of this section is to allow customers to perform production signing without requiring MEU to run on the signing server. The OEM may use MEU to debug/dummy sign first and then export the given manifest to a signing server for OEM proprietary signing flow.

7.1 Production Signing High-Level

After a component is signed with a debug (non-secure) key, or component is manifested yet not signed, the manifest may be exported to separate it from the main binary. The exported manifest can then be sent to the secure server for production signing.

The secure server will insert a production signature and public key hash into the manifest, which can then be imported back using MEU, to the original binary, creating the production signed component.



Note: The OEM "Production Key" is the key they wish to use for the given bin for platforms in the field. They may define this key to be pre-production or production per the needs (i.e. during R&D dedicate a "Pre-production" key and for launched platforms, use "Production" key.)

7.2 Export Manifests

Use the MEU –export function to export the manifest from binaries who need signatures added or changed. The manifest is exported to a

Production Signing

```
directory.
# meu -export -f <binary.bin> -o
<directory_containing_manifests>
```

If the binary includes multiple manifests, you must specify the index of the desired manifest, e.g.

```
# meu -export 0 -f <binary.bin> -o
<directory_containing_manifests>
```

If you do not supply an index or include `all` with the `-export` flag, MEU will output a list of all the manifests, including their indices:

More than one manifest was found in this file. Please provide a comma-separated list of the manifest indices you want to export. (ex. `-export "0,3,5"`) or specify `"all"` (ex. `-export "all"`)

The following manifests were detected:

Index	Offset	Size	Name (if available)
-------	--------	------	---------------------

0	0x000001130	0x000000D9C	FTPR.man
1	0x000053000	0x000000330	rot.key
2	0x000094058	0x000000378	RBEP.man
3	0x0000A1748	0x000001280	NFTP.man

Error 26: Failed to export manifest(s). Missing manifest indices list.

7.3 Manifest structures

In order to perform production signing on the OEM server, the OEM needs to re-sign the portion of the manifest, replace the signature and insert the production public key. This section details the manifest layout to enable this process.

7.3.1 Manifest Header

In order to use an alternate signing tool, the OEM needs to:

1. Sign the 'Signed Portion' of the manifests with the production signing key. The signed portion is the full manifest except for the signature and public key.
2. Change the Signature and Public Key section with the production signature and production public key used.

This means, the entire manifest binary must be hashed without the three crypto fields in the header: Public Key (offset 132, size 384), Exponent (offset 516, size 4) and Signature (offset 520, size 385). The hash must be performed using SHA-384, then be encrypted with PKCS #1-v1_5 to create the signature. Add the three crypto fields, key, exponent and signature, back into the manifest header.

No other fields in the manifest should be changed.

Structure of manifest header:

Entry Name	Offset	Size	Description
Type	0	4B	Must be 0x4
Length	4	4B	In Dwords. - Equals 161 for SHA-256, PKCSv1.5 version - 225 for SHA-384, SSA-PSS version.
Version	8	4B	- 0x10000 for SHA-256 PKCSv1.5 version - 0x21000 for SHA-384, SSA-PSS version
Flags	12	4B	Manifest Flags Bit 0 is PV bit flag: Intel components which are PV quality will have this bit set. This is ignored for OEM components. Bit 31 is debug flag: Optional use to indicate that manifest is debug signed. If set to true during debug signing, must be reverted to false on production signing facility.
Vendor	16	4B	Vendor ID
Date	20	4B	yyymmdd in BCD format
Size	24	4B	in Dwords size of the entire manifest. Maximum size is 2K DWORDS (8KB)
Header_id	28	4B	Magic number. Equals \$MN2 for this version
internal_data	32	4B	Must be 0x4 for all headers
version_major	36	2B	Major Version
version_minor	38	2B	Minor Version
version_hotfix	40	2B	Hotfix
version_build	42	2B	Build number
Svn	44	4B	Secure Version Number
meu_kit_version	48	8B	MEU Kit Version
meu_manifest_version	56	4B	Manifest Version - increased each fix/change that break backward compatibility. Last word is reserved for future use
reserved	60	60B	will be set to 0

Entry Name	Offset	Size	Description
modulus_size	120	4B	In DWORDs; 64 for pkcs 1.5-2048 , 96 for SSA-PSS - 3072
exponent_size	124	4B	In DWORDs; for pkcs 1.5:2048, and for SSA-PSS: 3072
Public Key	128	384B	Modulus in little endian format
Exponent	512	4B	Exponent in little endian format
Signature	516	384B	RSA signature of manifest extension in little endian. The signature is an PKCS #1-v1_5 of the entire manifest structure, including all extensions, and excluding the last 3 fields of the manifest header (Public Key, Exponent and Signature).

There may be multiple extensions after this manifest header making up the rest of the manifest binary.

7.3.2 Signed Package Info Extension

For authenticating the various platform firmware components such as cAVS, iUnit, ISH FW, etc. This structure will appear after manifest header for codepartitions.

Entry Name	Offset (Decimal)	Size	Description
Extension Type	0	4	= 15 for Signed Pkg Info Extension
Extension Length	4	4	In bytes; equals (52 + 52*n) for this version, where 'n' is the number of modules in the manifest
Package Name	8	4	Name of the package
Version Control Number (VCN)	12	4	The version control number (VCN) is incremented whenever a change is made to the FW that makes it incompatible from an update perspective with previously released versions of the FW
Usage Bitmap	16	16	Bitmap of usages depicted by this manifest, indicating which key is used to sign the manifest
SVN	32	4	SVN of this signed image
FW Type	36	1	Lowest 3 bits represent FW type: 0 - Reserved for future extensibility 1 - SPS The "normal" server SPS FW 2 - SPS EPO The endpoint-only mode server FW flavor

Entry Name	Offset (Decimal)	Size	Description
			3 - Client FW SKU (Corporate, Consumer, Lite, Slim) 4 - Reserved 5 - SPS server, Purley-R 6-15 Reserved
FW Sub Type	37	1	Lowest 3 bits represent FW Sub Type: For SPS FW type (1): 0 - Reserved. 1 - E5 SPS FW. 2 - E3 SPS FW. 3 - SPS SV FW. 4 - Ignition FW 5-255 - Reserved. For SPS EPO FW type (2): 0 - Reserved. 1 - SPS EPO FW. 2-255- Reserved. For Client FW type (3): 0 - Reserved. 1 - Client FW. 2 - HEDT (High End Desktop) FW. 3 - WS (WorkStation) FW. 4 - Converged Mobility FW. 5 - IOT FW 6 - Small Core Entry Level FW. 7-255 - Reserved.
Reserved	38	14	Must be 0
Module 0 Name	52	12	Character array; if name length is shorter than field size, the name is padded with 0 bytes.
Module 0 Type	64	1	0 - Process 1 - Shared Library 2 - Data 3 - Reserved
Module 0 Hash Algorithm	65	1	0 - Reserved 1 - SHA1 2 - SHA256

Entry Name	Offset (Decimal)	Size	Description
			3 – SHA384
Module 0 Hash Size	66	2	Size of Hash in bytes = N the Max size of (Module 0 Hash Algorithm = SHA384 -> 48 bytes)
Module 0 Metadata Size	68	4	Size of metadata file
Module 0 Metadata Hash	72	32	The SHA2 of the module metadata file
Module 0 Metadata 384 Hash	104	48	The SHA384 of the module metadata file
...			

7.3.3 Metadata extensions

Name	Offset	Size (bytes)	Description
Extension Type	0	4	= 10 for module attribute extension
Extension Length	4	4	In bytes; equals 56 for this version
Compression Type	8	1	0 – Uncompressed 1 – Huffman compressed 2 – LZMA compressed
Reserved	9	3	Must be 0
Uncompressed Size	12	4	Uncompressed image size, must be divisible by 4K
Compressed Size	16	4	Compressed image size. This is applicable for LZMA compressed modules only. For other modules, should be the same as "Uncompressed size" field.
Global Module Identifier	20	4	A globally unique identifier for the module. Bits 0-15: Module number, unique in the scope of the vendor: Bits 16-31: Vendor ID (PCI style)
Image hash	24	32	SHA2 Hash of uncompressed image

7.3.4 OEM Key Manifest

After Manifest Header for OEM KM, there will be Key Manifest Extension that is used for OEM KM.

Name	Offset	Size (bytes)	Description
Extension Type	0	4	= 14 for Key Manifest Extension
Extension Length	4	4	In bytes; equals $(36 + 68*n)$ for this version, where 'n' is the number of keys in the manifest
Key Manifest Type	8	4	0 – Reserved 1 – CSE Root of Trust (ROT) Key Manifest 2 – OEM Key Manifest
Key Manifest Security Version Number (KMSVN)	12	4	The security version number for the Key Manifest
OEM ID	16	2	OEM ID (assigned to Tier-A OEMs by Intel) 0 – Reserved/Not-Used Only least significant 16 bits are used in BXT.
Key Manifest ID	18	1	ID number of the Key Manifest. This is matched by the verifier against the value stored in the platform. This is typically used as a ODM ID – to enable an OEM to assign IDs to its various ODMs and generate Key Manifests specific to each ODM. Only least significant 4 bits are used in BXT.
Reserved	19	1	Must be 0
Reserved	20	16	Must be 0
Key 0 Usage	36	16	Bitmap of usages; allows for 128 usages. Bits 0-31 are allocated for Intel usages; bits 32-127 are allocated for OEM usages Bit 0-31: Reserved for Intel usage Bit 32: Reserved Bit 33: iUnit BootLoader Manifest Bit 34: iUnit Main FW Manifest Bit 35: cAVS Image #0 Manifest Bit 36: cAVS Image #1 Manifest Bit 37: Reserved Bit 38: OS Boot Loader Manifest Bit 39: OS Kernel manifest Bit 40: Reserved Bit 41: ISH manifest 1 (ISH Main)

Name	Offset	Size (bytes)	Description
			Bit 42: ISH manifest 2 (ISH BUP) Bit 43: OEM Debug Tokens Manifest Bit 44: Reserved Bit 45: Reserved Bit 46: Reserved Bit 47 - 127: Reserved for future use
Key 0 Reserved	52	16	
Key 0 Policy	68	1	BIT 0 : IPI. Relevant to ROT KM only. 1 – component is signed only by Intel 0 – component may be signed by OEM or Intel. When signed by OEM hash in OEM KM takes precedence. BIT 1..7 RESERVED
Key 0 Hash Algorithm	69	1	0 – Reserved 1 – SHA1 2 – SHA256 3 – SHA384
Key 0 Hash Size	70	2	Size of Hash in bytes = N BXT to support only SHA256. So N=32.
Key 0 Hash	72	N (32)	The hash of the key. BXT to support only SHA256. So N=32.
Key 0 384 Hash	104	48	The SHA384 hash of the key
...			

7.4 Import Manifest

Use the MEU -import function to import the signed manifest back into the binary. The signed manifest must be in a separate directory passed as an input parameter. If the binary supports multiple manifests (e.g. a full SPI binary), and the folder has multiple manifests, the command will be able to import them all back into the binary.

```
# meu.exe -import <directory_containing_manifests> -f
<input_binary.bin> -o <output_binary.bin>
```



8 Common Bring Up Issues and Troubleshooting Table

8.1 Common Bring Up Issues and Troubleshooting Table

Problem / Issue	Solution / Workaround
Intel MEU tool fails to run	<p>Confirm that the MEU_Config and template xml files are present in the same folder as the Intel MEU tool.</p> <p>Confirm that both files have been modified properly.</p>
Audio component fails to load although signed and entered into image as instructed	<p>Check in OEM KM, that the OEM audio component uses the cAVS0 key in OEM KM, not cAVS1.</p>
MFIT errors	<ol style="list-style-type: none"> 1. Check that public key hashes in OEM KM match the private keys used to sign the component. 2. Check that OEM public key hash in MFIT matches key used to sign OEM KM 3. Verify that codepartition, codepartitionmeta, oemkeymanifest and other relevant XML fields entered correctly 4. Ensure MEU version used is aligned with MFIT version (from same KIT)