

# **Cannon Lake / Coffee Lake Signing and Manifesting Guide for Intel® ME 12.0 FW**

**User Guide**

---

***Revision 1.2***

***November 2018***

**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 © 2018, Intel Corporation. All rights reserved.



# Contents

1	Introduction .....	6
1.1	OEM Key Manifest (OEM KM) .....	6
1.2	Goal .....	6
1.3	Pre-Requisites .....	6
1.4	Tools Used In This Document .....	7
1.5	Terminology .....	7
2	Intel® Manifest Extension Utility (Intel® MEU) .....	8
2.1	Usage .....	8
2.2	Examples .....	9
2.2.1	Generate Configuration XML Template .....	9
2.2.2	Generate Code partition XML .....	10
2.2.3	Generate Compressed and Signed Partition .....	10
3	OEM Component Signing High-Level .....	12
3.1	OEM Key Manifest Creation .....	12
3.2	Why the OEM KM is Important? .....	13
3.3	Creation process of OEM KM .....	13
4	Manifesting and Signing OEM Components in the IFWI Image .....	14
4.1	Creating a Signed IFWI Image .....	14
4.2	Opt-ing out of the OEM KM .....	15
5	Creating PKI Key Pairs .....	16
5.1	Introduction .....	16
5.2	Generating Key Pair for Signing .....	16
5.3	Creating the Public Key Hash: .....	16
5.3.1	Creating a Public Key Hash Using Intel® MEU .....	16
5.3.2	Creating Public Key Hash Manually .....	17
5.4	Key Security .....	18
6	Using Intel® MEU to Manifest & Sign .....	19
6.1	Introduction .....	19
6.2	Binary Manifesting Signing Overview .....	19
6.3	Intel® MEU Configuration .....	20
6.4	Intel® MEU Binlist .....	21
6.4.1	Bin-list usage .....	22
6.5	Intel® MEU Decomposition .....	22
6.6	Intel® MEU Re-sign .....	23
6.7	Different Binary Types Supported By Intel® MEU .....	23
6.7.1	ISH FW .....	24
6.7.2	IUnit / aDSP .....	25
6.7.3	Secure Tokens (OEM Unlock Tokens) .....	26
7	OEM Key Manifest .....	28
7.1	Introduction .....	28
7.2	Creation of Manifest .....	28
8	Add Components to Intel® FIT .....	33
8.1	Introduction .....	33



8.2	Include each production signed binary .....	33
8.3	Add the OEM Key Manifest .....	33
8.4	Add the Public Key Hash for OEM Key Manifest .....	33
8.5	Change the Key Manifest ID .....	34
9	Production Signing .....	35
9.1	Introduction .....	35
9.2	Production signing high-level .....	35
9.3	Export Manifests .....	36
9.4	Manifest structures .....	36
9.4.1	Manifest Header .....	38
9.4.2	Signed Package Info Extension .....	39
9.4.3	OEM Key Manifest .....	40
9.5	Import Manifest .....	41
10	Common Bring Up Issues and Troubleshooting Table .....	43
10.1	Common Bring Up Issues and Troubleshooting Table .....	43

## Figures

Figure 1: OEM KM Creation Process .....	12
Figure 2: Platform Chain of trust extended from Intel HW Root of Trust (ME ROM HW) .....	13
Figure 3: High Level Overview of Manifesting and Signing OEM Components in the IFWI Image .....	15
Figure 4: Schematic View of Manifesting and Signing Process .....	20
Figure 5: Intel MEU Configuration xml .....	21
Figure 6: Intel MEU list of Supported Binary Types .....	22
Figure 7: Code Partition xml .....	24
Figure 8: Code Partition Metadata xml .....	25
Figure 9: OEMUnlockToken xml .....	26
Figure 10: Default OEM Key Manifest XML .....	29
Figure 11: OEM Key Manifest with 1 key for multiple manfiests .....	30
Figure 12: OEM Key Manifest with 2 key for multiple manfiests .....	30
Figure 13: Entering OEM Key Manifest .....	33
Figure 14: Entering OEM Public Key Hash .....	34
Figure 15: Entering OEM Public Key Hash .....	34
Figure 16: Production Signing Flow for OEM FW Binaries .....	35
Figure 17: OEM KM Manifest Structure .....	37
Figure 18: Code Partition Manifest Structure .....	38

## Tables

Table 2-1. Options .....	8
Table 2. Components Recognized by Intel MEU, and How Key Manifests should Handle .....	31
Table 3: Manifest Header .....	38
Table 4: Signed Package Info Extension .....	39
Table 5: Key Manifest Extension .....	40



## Revision History

---

Revision Number	Description	Revision Date
0.5	Initial Release	May 2017
0.7	Updates: <ul style="list-style-type: none"><li>• Updated bin list image in Figure 6</li><li>• Clarified OEM KM use-case in Section 1.1</li><li>• Updated MEU tool usage per latest tool and help menu details in Section 2.1</li><li>• Added important note about permanent impact for not including OEM KM in section Sections 3.1 &amp; 8.3</li><li>• Clarified steps of opt-ing out of OEM KM in Section 4.2</li><li>• General typo &amp; clarification updates</li></ul>	July 2017
0.75	Updates: <ul style="list-style-type: none"><li>• Added Table 4 &amp; Section 9.4.2 for "Signed package Info Ext"</li><li>• Added Table 5 &amp; Section 9.4.3 for "Key Manifest Ext"</li><li>• Corrected Figure numbers in entire document</li></ul>	August 2017
0.9	Updates: <ul style="list-style-type: none"><li>• Added section 6.7.3 for Secure Tokens (OEM Unlock)</li><li>• Added details in intro and entire doc about scope including CFL platform using CNL PCH (with ME12 FW).</li></ul>	August 2017
0.91	Updates: <ul style="list-style-type: none"><li>• Source for LZMA tool</li></ul>	October 2017
1.0	Aligning revision number for PV	December 2017
1.1	Update Figure-5 in section 6.3 – New figure includes the removal of the default Signing Tool path.	March, 2018
1.2	Added LZMA version for ISH compressing	November 2018



# 1 Introduction

---

This document gives an overview of the process of manifesting and signing OEM components to be included in the IFWI image for Cannon Lake platforms as well as Coffee Lake (CNL PCH) platforms using ME12 FW.

OEMs are required to add manifests to components in the IFWI images if they wish to enable OEM Key Manifest (OEM KM) in order for ME FW to authenticate OEM Components. OEMs are encouraged to enable OEM KM usage as it extends platform root of trust extended by the Intel ME ROM as Intel's HW Root of Trust.

In any of these cases, OEMs must sign all components and include an OEM Key Manifest, as explained in this User Guide.

## 1.1 OEM Key Manifest (OEM KM)

The OEM KM is a key manifest containing the OEM's public key hashes for authenticating the OEM's components. The OEM KM is authenticated by the ME FW against the "OEM Public Key Hash" FPF that is provisioned during OEM/ODM manufacturing flow. Once the OEM KM binary is authenticated, the keys contained in it are subsequently used to authenticate various OEM components based on the key usages enabled. The OEM KM can be used to authenticate OEM signed components such as ISH FW, Audio FW, iUnit FW, OS kernel, and OS bootloader. There are three main use case for the OEM KM:

1. Dedicate separate key for each OEM component team (i.e. 1 key for ISH team, another for Audio FW, another for iUnit, etc.).
2. The key signing OEM KM cannot be revoked since its public key hash is the "OEM Public Key Hash" in FPF HW. But the keys used inside the OEM KM can be revoked if they are compromised allowing customers to easily push out updates for the OEM KM by increasing SVN/VCN number of the OEM KM itself and replacing the compromised key with a new one.
3. OEM outsourcing to Multiple ODMs using the same OEM KM keys. OEMs may differentiate between the ODMs by using different KM IDs while using the same keys for OEM KM.

## 1.2 Goal

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 production IFWI image

## 1.3 Pre-Requisites

The user should download and install the following kit.

- Latest Intel® ME FW kit: The kit can be downloaded from the following location:  
<https://platformsw.intel.com/>



- CNL/CFL Firmware Bring Up Guide: The overall platform bring-up procedure is described in this guide which can be found in the ME FW kit.
- CNL/CFL System Tools User Guide: The System Tools User Guide gives further detail on the usage of all the firmware manufacturing tools and is the definitive guide to the details of each tool's usage.

## 1.4 Tools Used In This Document

The following tools are referenced this document:

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

## 1.5 Terminology

Term	Description
Intel® FIT	Intel® Flash Image Tool
IBB	Initial Boot Block
IBBL	Initial Boot Block Loader
IFWI	Integrated Firmware Image (System FW Image on SPI)
ISH	Integrated Sensor Hub
OBB	OEM Boot Block
Intel® MEU	Intel® Manifest Extension Utility
SUT	System Under Test
CNL	Cannon Lake
CFL	Coffee Lake
OEM KM	OEM Key Manifest
EOM	End of Manufacturing
ROT KM	Root of Trust Key Manifest (containing Intel public key hashes to authenticate Intel signed FW components).



## 2 Intel® Manifest Extension Utility (Intel® MEU)

The Intel® Manifest Extension Utility (MEU) inputs a firmware binary created by a 3<sup>rd</sup> party and outputs an independent-updateable partition (IUP) that is compressed and signed. After completing this process the signed binary can be added to the SPI flash image using the Intel® FIT tool.

The Intel® Manifest Extension Utility (MEU) requires administrator privileges to run under Windows\* OS. The user needs to use the Run as Administrator option to open the CLI in Windows\* 7 64/32 bit and Windows\* 8.1 64/32 bit.

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 firmware binary. The LZMA tool is supplied with the ISH binary or may be downloaded from <http://7-zip.org/sdk.html>.
- Calls the signing infrastructure tool to sign the partition.

### 2.1 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] [-mndebug] [-st] [-stp] [-key] [-noverify] [-keyhash] [-resign]
        [-export] [-import]
```

**Table 2-1. Options**

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	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.





Option	Description
-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.
-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)
-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 directory.
-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.

## 2.2 Examples

### 2.2.1 Generate Configuration XML Template

This command will generate the configuration XML template file using MEU.

**Windows / WinPE:**

```
MEU.exe -gen meu_config
```

```
=====
=
```

```
Intel(R) Manifest Extension Utility. Version: 12.0.0.xxxx
```

```
Copyright (c) 2013 - 2017, Intel Corporation. All rights reserved.
```



7/12/2017 - 10:16:35 am

=====

Command Line: meu.exe -gen meu\_config

Log file written to meu.log

Saving XML ...

XML file written to meu\_config.xml

## 2.2.2 Generate Code partition XML

This command will generate the Code partition XML file using MEU.

### Windows / WinPE:

MEU.exe -gen CodePartition

=====

Intel(R) Manifest Extension Utility. Version: 12.0.0.xxxx  
Copyright (c) 2013 - 2017, Intel Corporation. All rights reserved.  
7/12/2017 - 10:16:35 am

=====

Command Line: meu.exe -gen CodePartition

Saving XML ...

XML file written to CodePartition.xml

## 2.2.3 Generate Compressed and Signed Partition

This command will create the compressed and signed partition using MEU.

### Windows / WinPE:

MEU.exe -f CodePartition.xml -o ISHC\_MEU.bin

=====

Intel(R) Manifest Extension Utility. Version: 12.0.0.xxxx  
Copyright (c) 2013 - 2017, Intel Corporation. All rights reserved.  
7/12/2017 - 10:16:35 am

=====

Command Line: meu.exe -f CodePartition.xml -o ISHC\_MEU.bin

Executing pre-build actions

Building objects

Processing attribute: CodePartition

Executing post-build actions



Full Flash image written to C:\...\ISHC\_MEU.bin

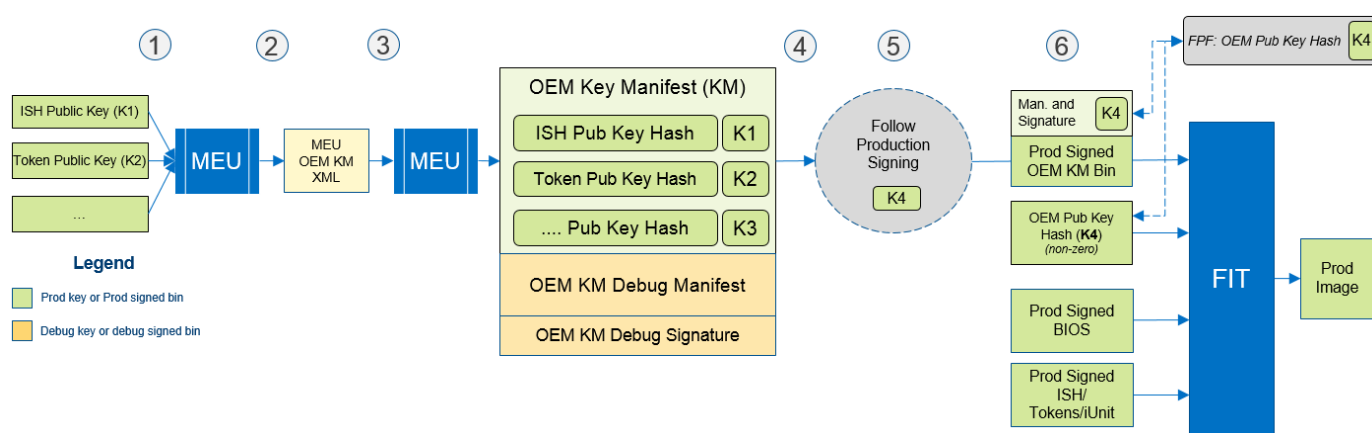


## 3 OEM Component Signing High-Level

This section describes high-level OEM component signing in the context of the OEM Key Manifest (KM).

### 3.1 OEM Key Manifest Creation

The diagram below demonstrates high-level flow of OEM KM creation.



**Figure 1: OEM KM Creation Process**

#### High-Level Procedure:

1. To authenticate OEM components using the OEM KM, input each of their public keys into MEU to produce public key hashes
2. Add the public key hash binaries to MEU XML
3. Input the MEU XML to MEU to generate the OEM KM binary
4. MEU outputs manifest, extensions, hash, debug signature in the OEM KM binary
5. Perform production signing on the OEM KM binary.
6. Input the production signed components into FIT (including the OEM KM signed binary) & OEM Public Key hash

**Important Note:** The OEM KM is optional. OEMs who do not wish to use the OEM KM may keep out the OEM KM binary. By excluding or including OEM KM binary, the given platform will be **permanently** set to require/not-require OEM KM per the configuration set in FIT. This choice will only be **permanently committed** to FPF HW at the time the platform undergoes closemnf/end-of-manufacturing process. This cannot be reversed after closemnf/EOM. This will be done by an FPF value called OEM\_KM\_Presence. This FPF value can be viewed by MEInfo.



## 3.2 Why the OEM KM is Important?

The Intel ME Authentication infrastructure is important to the OEMs platform because OEM can take advantage of Intel HW Root of Trust (ME ROM) to extend the chain of trust to the BIOS and even to the OS.



**Figure 2: Platform Chain of trust extended from Intel HW Root of Trust (ME ROM HW)**

## 3.3 Creation process of OEM KM

Please refer to this chapter: OEM Key Manifest.





## 4 Manifesting and Signing OEM Components in the IFWI Image

---

### 4.1 Creating a Signed IFWI Image

A high-level overview of creating a **signed** IFWI image using OEM components is described below. The key steps are:

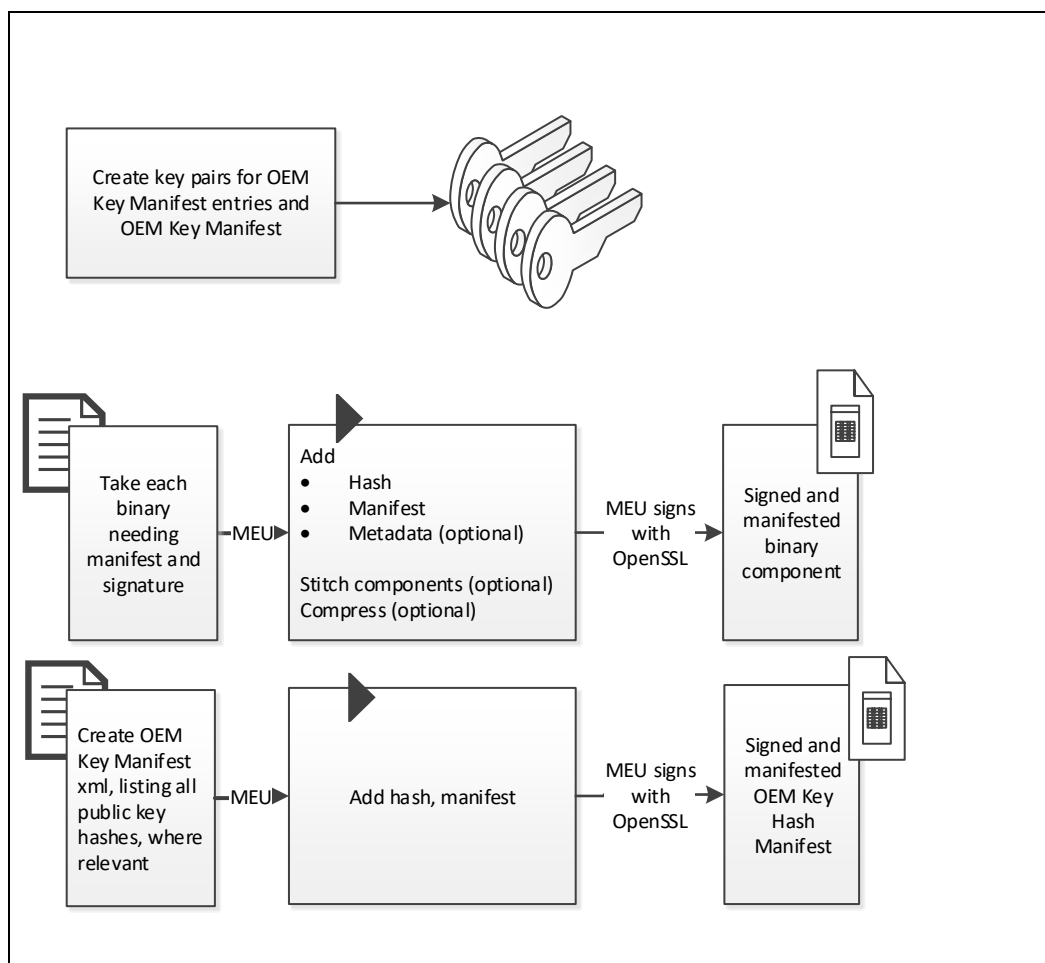
1. Generate PKI key pairs and the public key hash for:
  - a. Each entry in the OEM Key Manifest. These are enumerated in Section 7.2.
  - b. The OEM Key Manifest
2. Use the Intel® MEU tool to add to each binary a manifest, signature, and where relevant also add metadata, stitch and/or compress the binary.
3. Create an OEM Key Manifest<sup>1</sup>, including within it the public key hash of each of the created keys, and use the Intel MEU to manifest/sign it. Note: The order in which steps 2 and 3 are executed does not matter.
  - a. Signing the "OEM KM" binary may be done by performing the normal OEM proprietary production signing flow. Refer to the production signing section for more details.
4. Add each binary component to the Intel FIT.
5. Add to Intel FIT the OEM Key Manifest created in step 3.
6. Add to Intel FIT the public key hash for the key used to sign the OEM KM. At EOM (End of Manufacturing)/closemnf process, this value of the OEM Public Key hash will be committed/burned into the HW FPFs permanently.
7. For debug use-cases, you may add an OEM debug token to Intel FIT.

---

<sup>1</sup> OEM KM is optional. OEMs who do not wish to use OEM KM may keep OEM Public Key hash as zeros in FIT tool.



**Figure 3. High Level Overview of Manifesting and Signing OEM Components in the IFWI Image**



## 4.2 Opt-ing out of the OEM KM

OEMs who do not wish to utilize the OEM KM should:

1. Do not create nor include OEM KM binary into FIT. FIT will set an FPF value to indicate OEM KM is not present. This is FPF that will be permanently set in the FPF HW at time of EOM flow.
2. If using ISH, iUnit, or Audio FW, use Intel signed ISH, iUnit and Audio. Intel signed FW is authenticated by Intel ROT KM (Root of Trust Key Manifest) which is part of the Intel signed ME FW. Make sure to use pre-production ISH/Audio/iUnit with pre-production ME FW & production ISH/Audio/iUnit to make sure keys match and verified per keys in the appropriate ME FW.



## 5 Creating PKI Key Pairs

---

### 5.1 Introduction

If creating a signed IFWI image, you will need to create PKI key pairs, as well as the public key hash for:

1. Each entry in the OEM Key Manifest. See Creation of Manifest for full list of entries in the OEM KM.
2. The OEM Key Manifest

### 5.2 Generating Key Pair for Signing

The Intel tools are designed to work together with the open source OpenSSL tool (version 1.0.2b or higher), which generates key pairs in the RSA-2048 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, 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:  
`# openssl.exe genrsa -out privkey.pem 2048`

A public key can be extracted from the private key using:  
`# openssl.exe rsa -in privkey.pem -pubout -out pubkey.pem`

### 5.3 Creating the Public Key Hash:

A public key hash is a binary file containing the modulus and exponent of the public key in little endian format. You can create it using the Intel® MEU, or manually.

#### 5.3.1 Creating a Public Key Hash Using Intel® MEU

You can create the public key hash using the Intel® MEU in one of 3 different ways:

1. Extraction from an already signed binary:  
`# meu.exe -keyhash <output hashfile> -f <input.bin>`
2. Extraction from a public or private key in PEM format  
`# meu.exe -keyhash <output hashfile> -key <inputkey.pem>`
3. Creation when building or signing a binary





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

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

Here is an example of generating the public key hash from a signed binary:

```
# meu.exe -keyhash temp/hash -f iunp.bin
=====
=
Intel(R) Manifest Extension Utility. Version: 12.0.0.xxxx
Copyright (c) 2013 - 2017, Intel Corporation. All rights reserved.
7/12/2017 - 10:16:35 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.
-----
```

### 5.3.2 Creating Public Key Hash Manually

You can create a public key hash manually in one of two different ways:

1. Extraction from the public or private key:
  - a. Using OpenSSL, dump the key details:  
If using the public key:
 

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

If using the private key:

```
openssl.exe rsa -in private.pem -text -noout
```
  - b. Copy the modulus (excluding any leading bytes that are all 0s)
  - c. 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)
  - d. Paste the reverse byte modulus into a new file <new file> in a hex editor
  - e. Copy the exponent following the modulus into the new file (make sure it is little endian)

Hash the new file using



```
openssl.exe dgst -sha256 <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 -sha256 <new file>
```

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

## 5.4 Key Security

Although the same key may be used for signing each entry in the OEM Key Manifest, and indeed for signing the manifest itself, Intel recommends using separate key pairs for signing each component. Using the same key for signing multiple components is less secure, as if the key is compromised, the entire package is compromised.

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 3<sup>rd</sup> parties, they may be used to create and sign unofficial versions of the binaries which can then be loaded onto the platform.

Keys may be needed again if there is a need to re-sign a future version of a binary.

OEMs need to take special steps to ensure that the private keys are kept secure while allowing restricted/audited access to them for manifesting/signing components and building the image. 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 use a separate set of keys during the development process and creating production images. This will ensure that on production platforms only the production OEM Key Manifest with signatures for production components can be run.

### §



## **6      *Using Intel® MEU to Manifest & Sign***

---

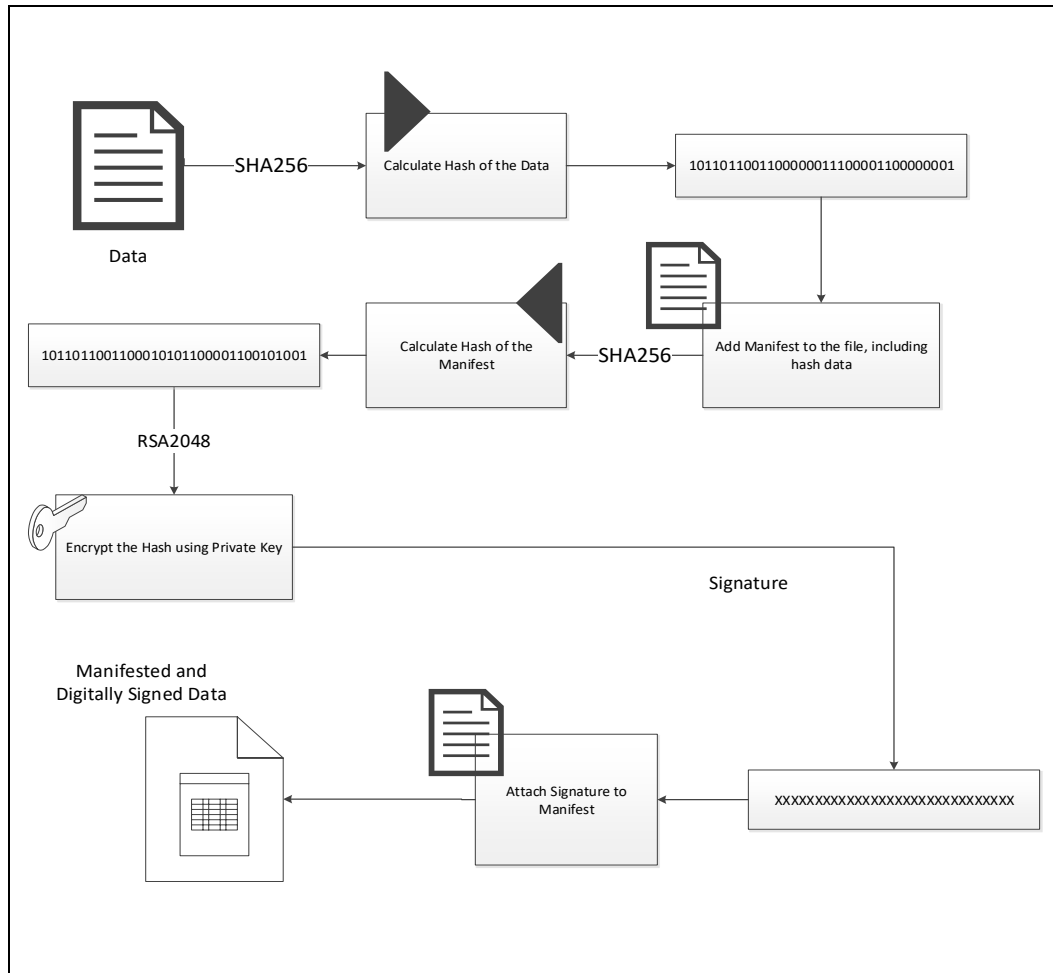
### **6.1      Introduction**

All of the components to be authenticated by keys in the OEM Key Manifest owned by an OEM are expected to have a signed manifest added to them.

### **6.2      Binary Manifesting Signing Overview**

Intel signing for CNL/CFL platforms employs an RSA 2048 public key infrastructure (PKI) mechanism to sign and verify components of the IFWI image. The private key is used to sign the image components as shown in Figure 2 below. The Intel MEU is used to create the manifests and interfaces with OpenSSL to add signatures to the manifest.

Figure 4. Schematic View of Manifesting and Signing Process



## 6.3 Intel® MEU Configuration

To get started using Intel MEU you must first configure the tool. To do this, run the following command:

```
# meu -gen meu_config
```

This will generate a default configuration xml file:



Figure 5. Intel MEU Configuration xml

```
<?xml version="1.0" encoding="UTF-8"?>
- <MeuConfig version="2.4">
  - <PathVars label="Path Variables">
    <WorkingDir label="$WorkingDir" help_text="Path for environment variable $WorkingDir" value="."/ />
    <SourceDir label="$SourceDir" help_text="Path for environment variable $SourceDir" value="."/ />
    <DestDir label="$DestDir" help_text="Path for environment variable $DestDir" value="."/ />
    <UserVar1 label="$UserVar1" help_text="Path for environment variable $UserVar1" value="."/ />
    <UserVar2 label="$UserVar2" help_text="Path for environment variable $UserVar2" value="."/ />
    <UserVar3 label="$UserVar3" help_text="Path for environment variable $UserVar3" value="."/ />
  </PathVars>
  - <SigningConfig label="Signing Configuration">
    <SigningTool label="Signing Tool" help_text="Select tool to be used for signing, or disable signing." value="OpenSSL"
      value_list="Disabled,,OpenSSL,,MobileSigningUtil" />
    <SigningToolPath label="Signing Tool Path" help_text="Provide the path to your signing tool executable, such as /usr/bin/openssl or C:\OpenSSL-
      Win32\bin\openssl.exe" value="" />
    <PrivateKeyPath label="Private Key Path" help_text="Path to private RSA key (in PEM format) to be used for signing. Key is required if using
      OpenSSL. If using MSU, and value is not-empty, this will override the key in the Signing Tool Config XML."
      value="$WorkingDir\private.pem" />
    <SigningToolXmlPath label="Signing Tool Config XML Path" help_text="Configuration XML template for MobileSigningUtil. Leave blank if not using
      MSU." value="" />
    <SigningToolExecPath label="Signing Tool Execution Path" help_text="Specify a directory from which the signing tool should be executed. This can
      be useful if relative paths are used in the Signing Tool Config XML. If no path is provided, the signing tool will be executed from the same
      directory as this tool was executed. Leave blank if not using MSU." value="" />
  </SigningConfig>
  - <CompressionConfig label="Compression Configuration">
    <LzmaToolPath label="LZMA Tool Path" help_text="Path to lzma tool executable." value="" />
  </CompressionConfig>
</MeuConfig>
```

If you will not be signing the manifests, then change the 'SigningTool' node to 'Disabled'.

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

If you will be signing the manifests, the xml should be edited to ensure the 'SigningToolPath' node correctly points to the OpenSSL executable file and that the path to the private key used for signing is correct. You are free to edit the other fields if appropriate.

## 6.4 Intel® MEU Binlist

Intel MEU supports manifesting and signing a large number of different file types. To see the full list, run the following:

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



Figure 6. Intel MEU list of Supported Binary Types

```
Intel(R) Manifest Extension Utility. Version: 12.0.0.1006
Copyright (c) 2013 - 2017, Intel Corporation. All rights reserved.
7/12/2017 - 10:54:40 am
=====

Command Line: 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
OEMKeyManifest       - OEM Key Manifest
OemUnlockToken       - OEM Unlock Token

Program terminated.
=====
```

### 6.4.1 Bin-list usage

Each of the items in the above binlist is supported by MEU for manifesting & signing. For each file that needs to be manifested and signed you use the Intel MEU to generate an xml for that file type and then edit the xml to ensure the data is correct – in particular:

1. Update xml with targeted FW usage
2. Update xml with targeted FW path

You then call the MEU with the edited xml as input, and pass in the name of the required output file, it will create the manifested and signed output file:

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

It is recommended practice to sign each file with a different private key. An easy way to do this is to use the configuration xml without changing it and override the private key used for signing on the command line (otherwise the private key in the MEU config would be used to sign all the components):

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

## 6.5 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 while providing an xml detailing the decomposition. This xml can later be used as input to the Intel® MEU to recreate the full binary with manifest and signature. The `-decomp` command also requires the binary type as its first parameter. So, for example, to decompose an OEM Key Manifest binary, you can call:

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



## 6.6 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>
```

It is only necessary to override the private key for signing (as in the example) if the key is different than that defined in the default Intel® MEU configuration xml.

Some binaries – such as full IFWI images, include multiple manifests. When calling the `-resign` option on such binaries, you need 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>
```

## 6.7 Different Binary Types Supported By Intel® MEU

Intel MEU is able to add manifests and sign several types of files, as enumerated below.

**Note:** All binaries provided by Intel will have a manifest and signature. OEMs do not need any further processing on these binaries.



## 6.7.1 ISH FW

The ISH FW binary is regarded as a 'code partition' by the Intel® MEU. Intel signed ISH FW is authenticated by the Intel® ME FW, but if the OEM wishes to have their own custom ISH FW instead of the Intel signed version, the OEM may do so by customizing the FW then signing with their private key and including the public key hash in the OEM KM. To sign such FW, the OEM needs to generate the code partition xml template using the following command:

```
# meu -gen CodePartition
```

The xml generated will need to be edited to enter version information about the code partition, as well as the path to the binary.

If compression is required, as with ISH, the path to the LZMA compression file must be entered. The current supported version of LZMA is 9.2, and it can be downloaded independently from the web.

Figure 7. Code Partition xml

```
<?xml version="1.0" encoding="UTF-8"?>
- <CodePartition version="2.4">
  <Name help_text="Name to use in the output binary's directory. Maximum length is 4 characters." value="ISHC"/>
  <Length 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." value="0x0"/>
  <Usage help_text="Indicates the type of data contained in this binary. This value is used during signature verification to validate the public key." value="IshManifest"
    value_list="CseBupManifest,,CseMainManifest,,PmcManifest,,UsbTypeCIOMManifest,,UsbTypeCMGManifest,,UsbTypeCTBTManifest,,IshManifest"/>
  <VendorId value="0x0000"/>
  <InstanceId value="0x1"/>
  <PartitionFlags value="0x00000000"/>
  <PartitionVersion value="0x10000000"/>
  <VersionControlNumber value="0x00000000"/>
  <SecurityVersionNumber value="0x00000000"/>
  <VersionMajor help_text="Used to manually set the Major Version field in the manifest" value="0x0" label="Version Major"/>
  <VersionMinor help_text="Used to manually set the Minor Version field in the manifest" value="0x0" label="Version Minor"/>
  <VersionHotfix help_text="Used to manually set the Hotfix Version field in the manifest" value="0x0" label="Version Hotfix"/>
  <VersionBuild help_text="Used to manually set the Build Version field in the manifest" value="0x0" label="Version Build"/>
  - <VersionExtraction>
    <Enabled 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." value="false" value_list="true,,false"/>
    <InputFile help_text="Binary file from which to extract the version details." value="" value_list=""/>
    <VersionMajorByte0Offset help_text="Offset of Major Version number's LSB in InputFile." value="0"/>
    <VersionMajorByte1Offset help_text="Offset of Major Version number's MSB in InputFile." value="0"/>
    <VersionMinorByte0Offset help_text="Offset of Minor Version number's LSB in InputFile." value="0"/>
    <VersionMinorByte1Offset help_text="Offset of Minor Version number's MSB in InputFile." value="0"/>
    <VersionHotfixByte0Offset help_text="Offset of Hotfix Version number's LSB in InputFile." value="0"/>
    <VersionHotfixByte1Offset help_text="Offset of Hotfix Version number's MSB in InputFile." value="0"/>
    <VersionBuildByte0Offset help_text="Offset of Build Version number's LSB in InputFile." value="0"/>
    <VersionBuildByte1Offset help_text="Offset of Build Version number's MSB in InputFile." value="0"/>
  </VersionExtraction>
  - <CPModules>
    - <CPDataModule name="ish_main">
      <InputFile help_text="Path to binary file to load for this module's data." value="ish_main.bin"/>
      <CompressionType help_text="Select compression type for this module." value="LZMA"
        value_list="NOT_COMPRESSED,,LZMA"/>
      <ProcessId value="0xf6"/>
    </CPDataModule>
  </CPModules>
</CodePartition>
```

Once the Code Partition 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 the Code Partition with the private key created for this purpose.

```
# meu.exe -f <CodePartition.xml> -o <ISH.bin> -key<privatekey.pem>
```

It is only necessary to override the private key for signing (as in the example) if the key is different than that defined in the default Intel MEU configuration xml.





### 6.7.2 IUnit / aDSP

The IUnit and aDSP (Audio) FW binaries are regarded as 'code partition metadata' by the Intel MEU. Intel signed iUnit & aDSP (Audio) FW is authenticated by the Intel ME FW, but if the OEM wishes to substitute their own custom iUnit/aDSP FW the OEM may do so by customizing the FW, signing it with their private key and including the public key hash in the OEM KM. To sign such FW, the OEM needs to generate xml for it using the following command:

```
# meu -gen CodePartitionMeta
```

The xml generated will need to be edited to enter the path to the binary and the path to a metadata binary file.

### Figure 8. Code Partition Metadata xml

```
<?xml version="1.0" encoding="UTF-8"?>
<CodePartitionMeta version="2.4">
  <Name help_text="Name to use in the output binary's directory. Maximum length is 4 characters." value="IUNP"/>
  <Length 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." value="0x0"/>
  <Usage help_text="Indicates the type of data contained in this binary. This value is used during signature verification to validate the public key." value="" value_list="CseBupManifest,,CseMainManifest,,PmcManifest,,BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,VendorId help_text="32-bit Vendor ID value. (ex. Intel=0x8086)" value="0x0000"/>
  <InstanceId value="0x1"/>
  <PartitionFlags value="0x00000000"/>
  <PartitionVersion value="0x10000000"/>
  <VersionControlNumber value="0x00000000"/>
  <SecurityVersionNumber value="0x00000000"/>
  <VersionMajor help_text="Used to manually set the Major Version field in the manifest" value="0x0" label="Version Major"/>
  <VersionMinor help_text="Used to manually set the Minor Version field in the manifest" value="0x0" label="Version Minor"/>
  <VersionHotfix help_text="Used to manually set the Hotfix Version field in the manifest" value="0x0" label="Version Hotfix"/>
  <VersionBuild help_text="Used to manually set the Build Version field in the manifest" value="0x0" label="Version Build"/>
  - <VersionExtraction>
    <Enabled 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." value="false" value_list="true,false"/>
    <InputFile help_text="Binary file from which to extract the version details." value="/"/>
    <VersionMajorByte0Offset help_text="Offset of Major Version number's LSB in InputFile." value="0"/>
    <VersionMajorByte1Offset help_text="Offset of Major Version number's MSB in InputFile." value="0"/>
    <VersionMinorByte0Offset help_text="Offset of Minor Version number's LSB in InputFile." value="0"/>
    <VersionMinorByte1Offset help_text="Offset of Minor Version number's MSB in InputFile." value="0"/>
    <VersionHotfixByte0Offset help_text="Offset of Hotfix Version number's LSB in InputFile." value="0"/>
    <VersionHotfixByte1Offset help_text="Offset of Hotfix Version number's MSB in InputFile." value="0"/>
    <VersionBuildByte0Offset help_text="Offset of Build Version number's LSB in InputFile." value="0"/>
    <VersionBuildByte1Offset help_text="Offset of Build Version number's MSB in InputFile." value="0"/>
  </VersionExtraction>
  - <CodePartitionMetadata>
    <Name help_text="Name to use as metadata filename in output binary. Maximum length is 12 characters." value="iunit.met"/>
    <InputFile help_text="Local path to metadata binary file" value="iunit_met.bin"/>
  </CodePartitionMetadata>
  - <CPMModules>
    - <CPMDDataModule enabled="true" name="iunit">
      <InputFile help_text="Path to binary file to load for this module's data." value="iunit.bin"/>
      <CompressionType help_text="Select compression type for this module." value="NOT_COMPRESSED" value_list="NOT_COMPRESSED,,LZMA"/>
    </CPMDDataModule>
  </CPMModules>
</CodePartitionMeta>
```

Once the Code Partition Metadata 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.

```
# meu.exe -f <CodePartitionMeta.xml> -o <IUnit.bin> -key  
<privatekey.pem>
```



It is only necessary to override the private key for signing (as in the example) if the key is different than that defined in the default Intel MEU configuration xml.

### 6.7.3 Secure Tokens (OEM Unlock Tokens)

The OEMUnlockToken binary is authenticated by the Intel ME 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
```

The xml generated will need to be edited to enter the path to the part ID binary

Figure 9. OEMUnlockToken xml

```
<?xml version="1.0" encoding="UTF-8"?>
- <OemUnlockToken version="2.4">
  <ExpirationSeconds help_text="Time from Part ID generation to Token expiration (in seconds)."
    value="0x00278D00"/>
  <PartIdsPath help_text="Path to directory containing Part ID binaries." value=""/>
  - <TokenFlags>
    <PartRestricted value="Yes" value_list="Yes,,No"/>
    <AntiReplayProtected value="Yes" value_list="Yes,,No"/>
    <TimeLimited value="Yes" value_list="Yes,,No"/>
  </TokenFlags>
  - <TokenKnobs>
    <OemUnlockKnob value="OemUnlockEnabled"
      value_list="OemUnlockDisabled,,OemUnlockEnabled"/>
    <IshGdbDebugKnob value="IshGdbSupportEnabled"
      value_list="IshGdbSupportDisabled,,IshGdbSupportEnabled"/>
    <AllowVisaOverrideKnob value="VisaOverrideDisabled"
      value_list="VisaOverrideDisabled,,VisaOverrideEnabled"/>
    <DisableBiosSecureBootKnob value="SecureBootEnforced"
      value_list="SecureBootEnforced,,AllowRnDKeys,,SecureBootDisabled"/>
    <DisableAudioFwAuthenticationKnob value="AuthenticationEnforced"
      value_list="AuthenticationEnforced,,AllowRnDKeys,,AuthenticationDisabled"/>
    <DisableIshFwAuthenticationKnob value="AuthenticationEnforced"
      value_list="AuthenticationEnforced,,AllowRnDKeys,,AuthenticationDisabled"/>
    <DisableIunitFwAuthenticationKnob value="AuthenticationEnforced"
      value_list="AuthenticationEnforced,,AllowRnDKeys,,AuthenticationDisabled"/>
  </TokenKnobs>
</OemUnlockToken>
```

There are multiple flags that can now be set for the token as following:

In the **TokenFlags** tag, you can set following values to yes

- **PartRestricted:** This means that the token can be used on any platform whose token key hash matches that of the token, and tied to a particular platform ID when value is set to yes.
- **Anti-Replay Protected.** Anti-Replay protection stops a token being re-used on the same device after new token is created for device. This option is only relevant for tokens tied to a particular platform ID.
- **TimeLimited.** This means that the token has time limit. Anti-Replay Protected must be set for token with time expiration, because otherwise you can re-use the token after RTC clear.

It is recommended to use to secure token with time expiration and Anti-reply 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 set generated earlier for the given platform

In the TokenKnobs section, you can 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. For CNP it will enable debug interfaces to ISH and Audio
ISH GDB Debug	Enable ISH GDB support

**Note:** VISA override, DisableSecureBootknob, DisableIshFwAuthenticationKnob, DisableIunitFwAuthenticationKnob, DisableAudioFwAuthenticationKnob are not supported with OEM Secure Token and should be set to disabled

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.

```
# meu.exe -f <OEMUnlockToken.xml> -o <OEMtoken.bin> -key  
<privatekey.pem>
```

It is only necessary to override the private key for signing (as in the example) if the key is different than that defined in the default Intel MEU configuration xml.

## §



## 7 OEM Key Manifest

---

### 7.1 Introduction

The OEM Key Manifest is the central part of the entire signing mechanism. It lists the public key hashes of all the OEM-created binaries within the IFWI as well as other binaries and manifests that can be loaded at a later date (such as audio and camera binaries, OS Kernel and OS Boot loader, and secure tokens).

If the IFWI image will not be signed, the OEM can skip the creation of an OEM Key Manifest.

The OEM Key Manifest itself, once created, is signed with a key whose public key hash will be entered into Intel FIT. When the platform manufacture is complete, this public key hash will be burned into a fuse (FPF) that can never be changed. Thus we create a secure verification mechanism: firmware is able to verify that the OEM Key Manifest on the platform is the same one whose hash is burned into a hardware fuse, and each hash within the manifest allows firmware to verify the binary or manifest components it plans to load.

#### **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 a new image needs to be burned onto the platform (e.g. via flash gang programmer), it must be signed with this key.

### 7.2 Creation of Manifest

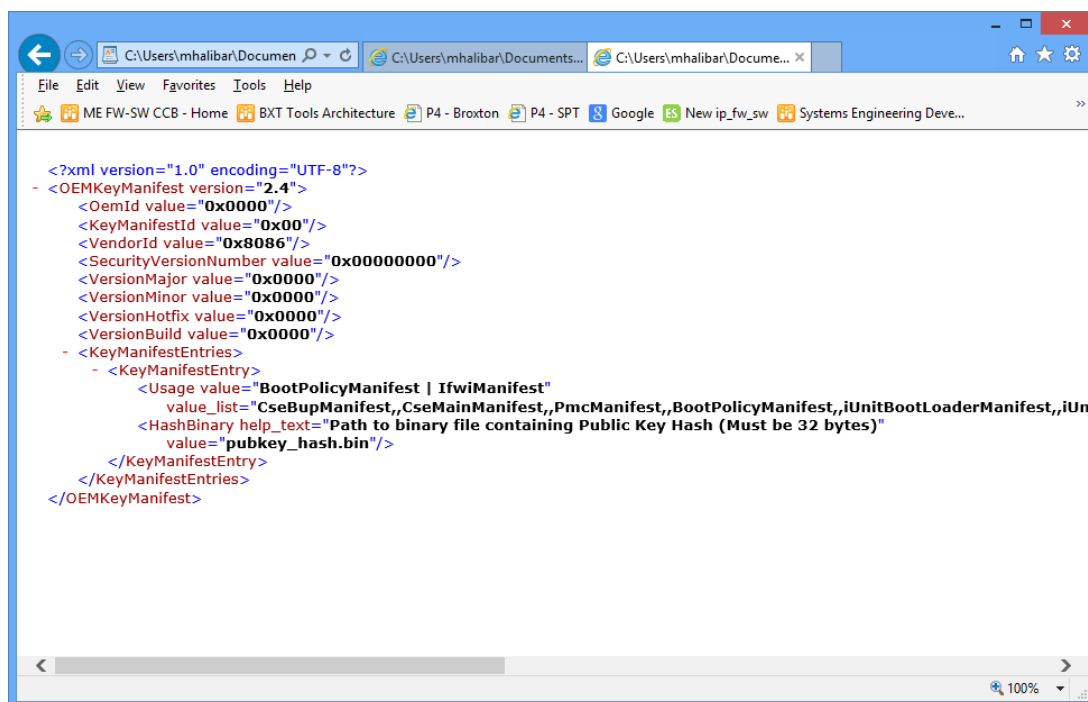
The manifest file xml template can be generated using the following command:

```
# meu -gen OEMKeyManifest
```

This generates an xml template with a single KeyManifestEntry node, which lists the file type, and the path to its public key hash.



Figure 10. Default OEM Key Manifest XML



The KeyManifestId field must not be left with its default value of 0x1 (must be given some non-zero value). It is critical that the matching field in FIT is also changed to match the non-zero value, as this field will be burned into an FPF and used to validate the OEM Key Manifest on platform boot.

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 wants to have

- IshManifest, iUnitBootLoaderManifest & iUnitMainFwManifest with key 1

It would appear as follows:



Figure 11. OEM Key Manifest with 1 key for multiple manifests

```
<?xml version="1.0" encoding="UTF-8"?>
- <OEMKeyManifest version="2.4">
  <OemId value="0x0000"/>
  <KeyManifestId value="0x1"/>
  <VendorId value="0x8086"/>
  <SecurityVersionNumber value="0x00000000"/>
  <VersionMajor value="0x0000"/>
  <VersionMinor value="0x0000"/>
  <VersionHotfix value="0x0000"/>
  <VersionBuild value="0x0000"/>
  - <KeyManifestEntries>
    - <KeyManifestEntry>
      <Usage value="IshManifest | iUnitBootLoaderManifest | iUnitMainFwManifest"
        value_list="BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0M
        <HashBinary value="pubkey_hash1.bin" help_text="Path to binary file containing Public Key Hash
          (Must be 32 bytes)"/>
      </KeyManifestEntry>
    </KeyManifestEntries>
  </OEMKeyManifest>
```

If the OEM Key Manifest wants to have

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

It would appear as follows:

Figure 12. OEM Key Manifest with 2 key for multiple manifests

```
<?xml version="1.0" encoding="UTF-8"?>
- <OEMKeyManifest version="2.4">
  <OemId value="0x0000"/>
  <KeyManifestId value="0x1"/>
  <VendorId value="0x8086"/>
  <SecurityVersionNumber value="0x00000000"/>
  <VersionMajor value="0x0000"/>
  <VersionMinor value="0x0000"/>
  <VersionHotfix value="0x0000"/>
  <VersionBuild value="0x0000"/>
  - <KeyManifestEntries>
    - <KeyManifestEntry>
      <Usage value="IshManifest"
        value_list="BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,cAvsImage0Manifest"
        <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"
        <HashBinary value="pubkey_hash2.bin" help_text="Path to binary file containing Public Key Hash
          (Must be 32 bytes)"/>
      </KeyManifestEntry>
    </KeyManifestEntries>
  </OEMKeyManifest>
```

The file types enumerated in the OEM Key Manifest for which key hashes can be entered are:



Table 2. Components Recognized by Intel MEU, and How Key Manifests should Handle

Manifest	POR CNL/CFL?	Description	Usage
<i>iUnitBootLoaderManifest</i>	Y	Camera firmware boot loader	If OEM wishes to customize this FW, they may sign customized FW and include its public key hash in OEM KM. Otherwise, Intel signed version of this FW is authenticated by ME FW without any need to include any keys by OEMs.
<i>iUnitMainFwManifest</i>	Y	Camera main firmware	
<i>IshManifest</i>	Y	Integrated Sensor Hub main firmware.	
<i>cAvsImage0Manifest</i>	Y	Audio (aDSP) firmware 0	
<i>cAvsImage1Manifest</i>	Y	Audio (aDSP) firmware 1	
<i>OemDebugManifest</i>	Y	OEM Debug token	
<i>OsBootLoaderManifest</i>	Y	OS Boot loader	
<i>OsKernelManifest</i>	Y	OS Kernel	
<i>SilentLakeVmmManifest</i>	N	Silent Lake manifest	
<i>IfwiManifest</i>	N	For small core IFWI2.0 capsule update via BIOS	
<i>BootPolicyManifest</i>	N	For Small core Boot Guard 2.0 Intel TXE verified boot.	
<i>OemSmipManifest</i>	N	Small core (APL/GLK) SMIP includes many of the settings defined in Intel FIT	

Not every hash listed in the table above is mandatory – for example, if no aDSP audio firmware is planned to be supported, the manifest may omit the audio entries. In such a case audio firmware would fail to be loaded by ME, if attempted. Likewise, if the OEM is not using the Intel's APIs to verify the OS kernel and manifest, then the respective hashes do not need to be included in the OEM Key Manifest. If the OEM does not plan to support Secure OEM debug Tokens, then the token hashes do not need to be included.

Once the OEM Key Manifest xml has been edited to include all the required hashes, the MEU can be run with the xml as input to manifest and sign the with the private key created for this purpose (private key to be used here will require to have its public



key hash set in the "OEM Pub Key Hash" FPF in FIT. This hash will be permanently committed to the FPF HW at EOM/Closemnf):

```
# meu.exe -f <OEMKeyManifest.xml> -o < OEMKeyManifest.bin> -key  
<privatekey.pem>
```

It is only necessary to override the private key for signing (as in the example) if the key is different than that defined in the default Intel MEU configuration xml.

## §





## 8 Add Components to Intel® FIT

### 8.1 Introduction

Intel FIT is a tool provided to OEMs to stitch together multiple binary files, configuration data and 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 & FIT functionalities are detailed in the Cannon Lake - Intel® ME Firmware Bring-Up Guide & System Tools User Guide.

### 8.2 Include each production signed binary

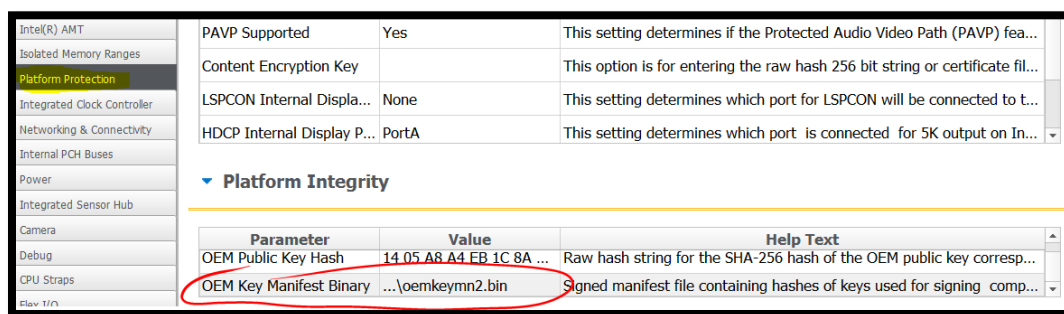
FIT includes input fields allowing the input of binary files. Most are available in the Flash Layout tab.

### 8.3 Add the OEM Key Manifest

Add the signed OEM KM binary into FIT.

**Important Note:** The OEM KM is optional. OEMs who do not wish to use the OEM KM may keep out the OEM KM binary. By excluding or including OEM KM binary, the given platform will be **permanently** set to require/not-require OEM KM per the configuration set in FIT. This choice will only be **permanently committed** to FPF HW at the time the platform undergoes closenmf/end-of-manufacturing process. This cannot be reversed after closenmf/EOM. This will be done by an FPF value called OEM\_KM\_Presence. This FPF value can be viewed by MEInfo.

Figure 13. Entering OEM Key Manifest



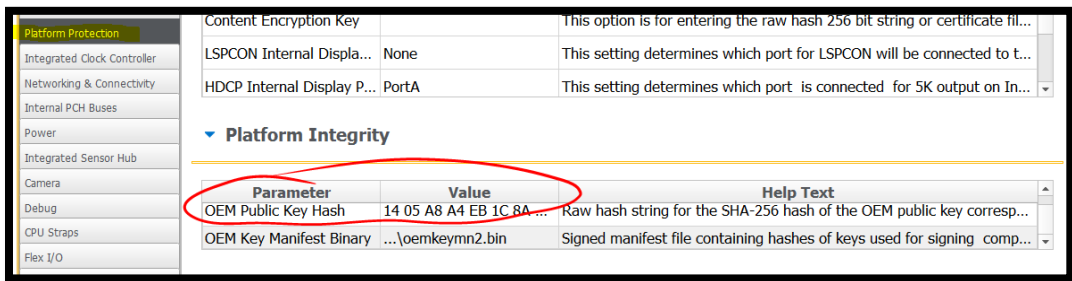
### 8.4 Add the Public Key Hash for OEM Key Manifest

Add to Intel FIT the public key hash for the OEM Key Manifest. This field is available in the Platform Protection tab.



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.

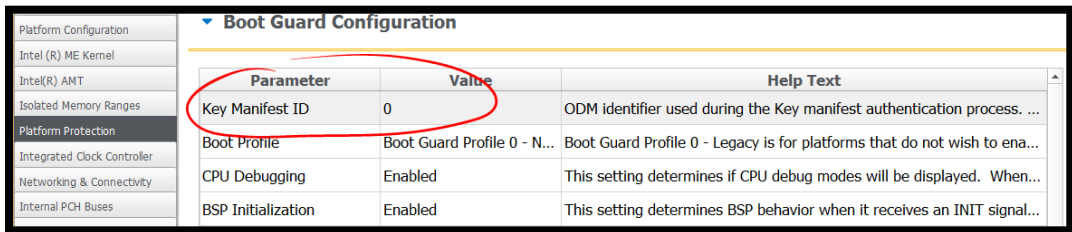
Figure 14. Entering OEM Public Key Hash



## 8.5 Change the Key Manifest ID

The Key Manifest ID field must be changed from 0x0 to match the value set in the OEM Key Manifest.

Figure 15. Entering OEM Public Key Hash



§

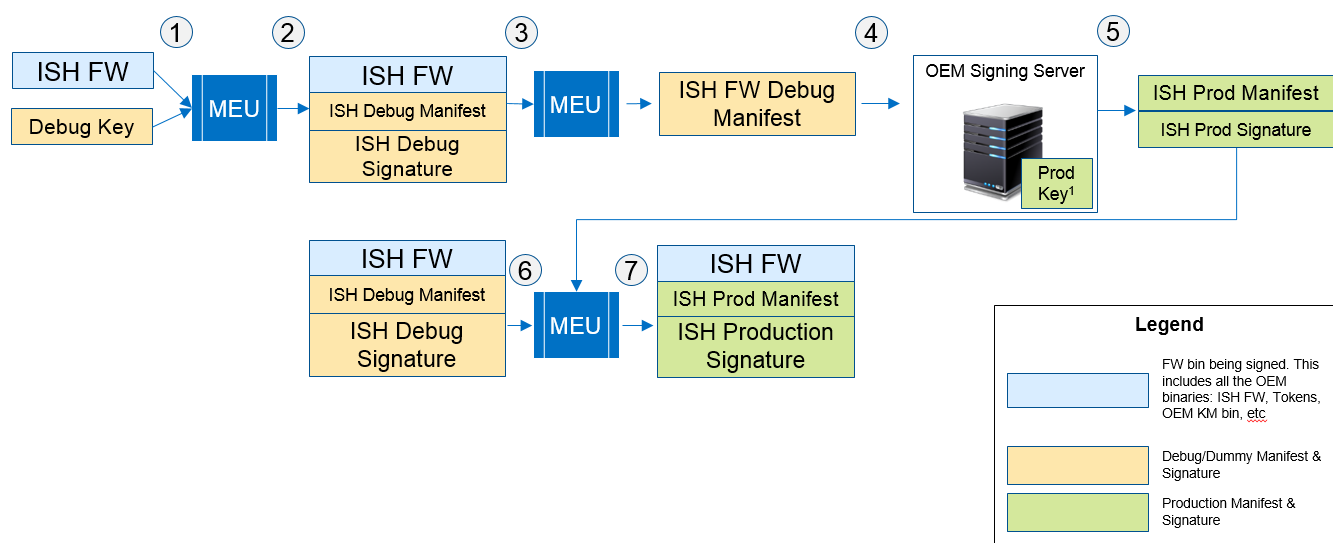
## 9 Production Signing

### 9.1 Introduction

Some OEMs will have already existing signing tools and systems and will want to use Intel® MEU together with them without having to integrate with OpenSSL. In order to do that the OEM needs to use the MEU to create the manifests required and then perform production signing separately.

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 signing first and then export the given manifest to a signing server to perform the OEM proprietary signing flow.

### 9.2 Production signing high-level



**Figure 16: Production Signing Flow for OEM FW Binaries**

#### High-level production signing process:

1. Pass the FW binary to be manifested & signed by the MEU (integrating with OpenSSL)
2. MEU adds manifest, extensions, hash, debug signature
3. Use MEU to extract debug signed manifest via export function
4. Pass the debug signed manifest to OEM signing infrastructure
5. Remove the debug signature + debug public key + sign the exported manifest with OEM signing infrastructure by inserting the production manifest + production public key
6. Pass production signed manifest and debug signed manifest+binary to MEU
7. MEU swaps the production signed manifest in place of debug signed manifest



**Note:** The OEM "Production Key" is the key the 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.)

## 9.3 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 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	0x0000001130	0x0000000D9C	FTPR.man
1	0x0000053000	0x0000000330	rot.key
2	0x0000094058	0x0000000378	RBEP.man
3	0x00000A1748	0x0000001280	NFTP.man

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

## 9.4 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.

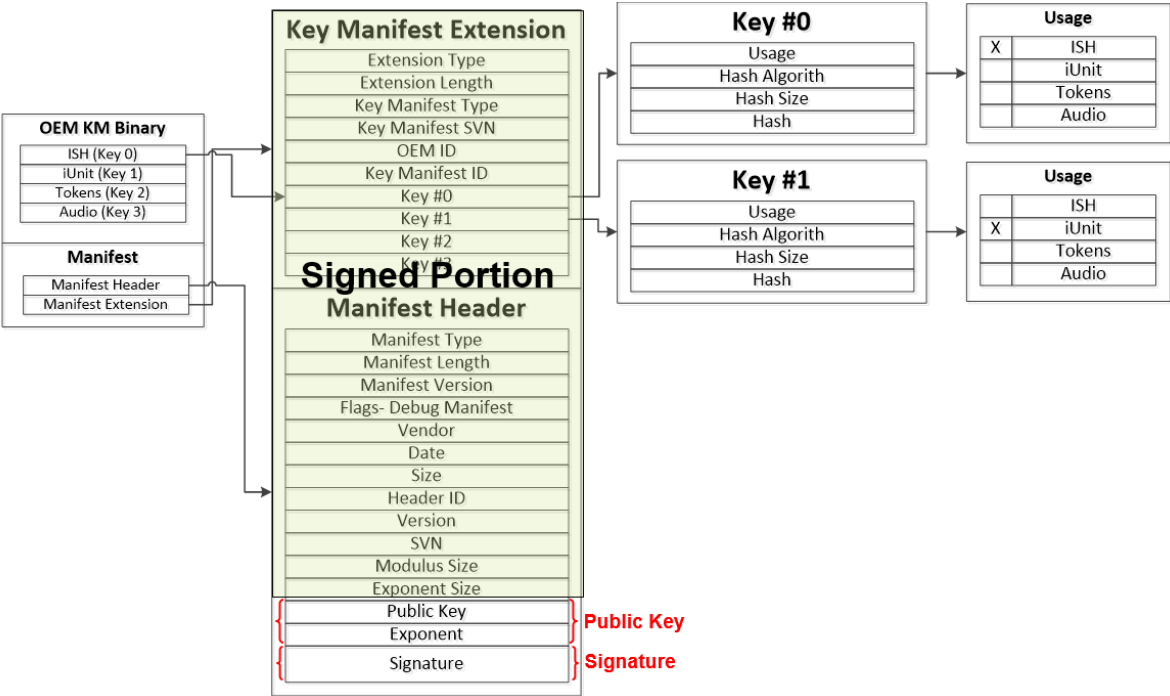
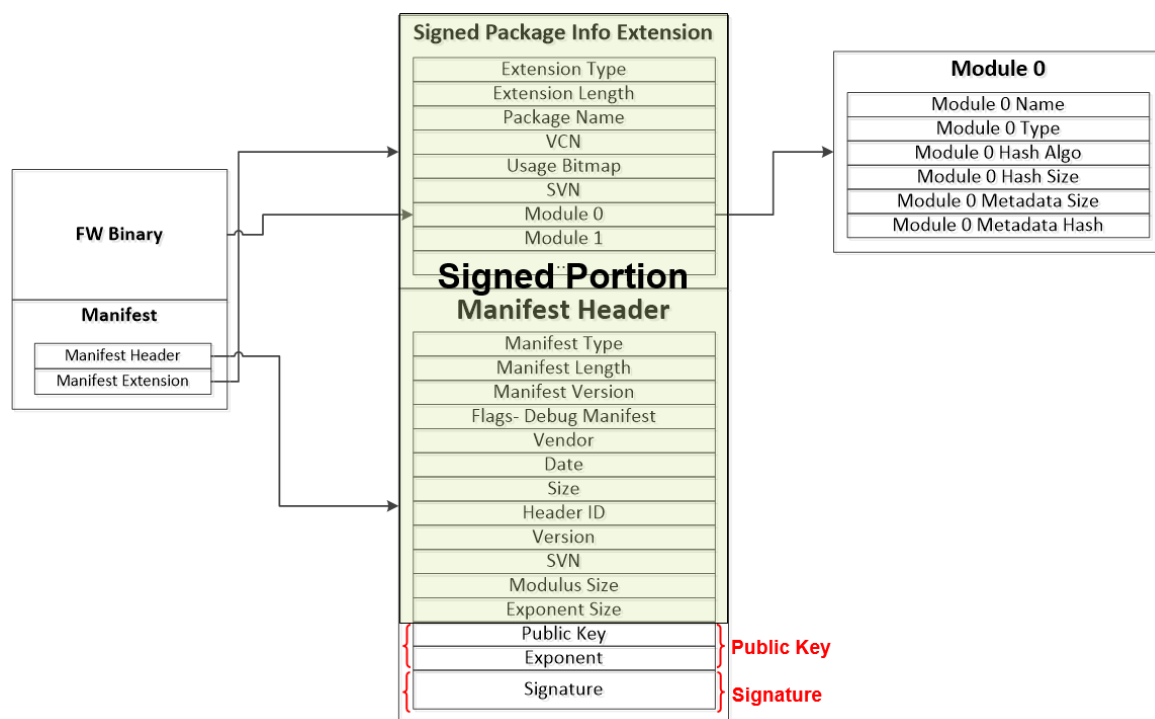


Figure 17: OEM KM Manifest Structure



**Figure 18: Code Partition Manifest Structure**

## 9.4.1 Manifest Header

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

1. Take the "Signed Portion" section of the above shown manifests and re-sign with the production signing key.
2. Change the "Signature" and "Public Key" section with the production signature and production public key used.

Structure of manifest header:

**Table 3: Manifest Header**

Name	Offset (Dec)	Offset (Hex)	Size (bytes)	Description
Header type	0	0	4	Must be 0x4
Header Length	4	4	4	In DWORDs; equals 161 for this version
Header Version	8	8	4	0x10000 for this version
Flags	12	C	4	Bit 31: Debug Manifest (manifest is debug signed, not production signed) Bits 0-30: reserved, must be 0
Vendor	16	10	4	0x8086 for Intel



Name	Offset (Dec)	Offset (Hex)	Size (bytes)	Description
Date	20	14	4	yyyymmdd in BCD format
Size	24	18	4	In DWORDs, size of entire manifest (header + extensions). Maximum size is 2K DWORDs (8KB).
Header ID	28	1C	4	Magic number. Equals "\$MN2" for this version
Reserved	32	20	4	Must be 0
Version	36	24	8	Major, minor, hotfix, build
Security Version Number	44	2C	4	SVN, least significant byte used to derive keys
Reserved	48	30	8	Must be 0
Reserved	56	38	64	Must be 0
Modulus Size	120	78	4	In DWORDs; 64 for pkcs 1.5-2048
Exponent Size	124	7C	4	In DWORDs; 1 for pkcs 1.5-2048
Public Key	128	80	256	Modulus in little endian format
Exponent	384	180	4	Exponent in little endian format
Signature	388	184	256	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.

The entire manifest binary must be hashed using SHA-256, except for the 3 'crypto' fields in the header: Public Key (offset 128, size 256), Exponent (offset 384, size 4) and Signature (offset 388, size 256). The hash must then be encrypted with PKCS #1-v1\_5 to create the signature followed by the 3 'crypto' fields in the manifest header populated with the key, exponent and signature.

No other fields in the manifest should be changed.

## 9.4.2 Signed Package Info Extension

For authenticating the various platform firmware components such as aDSP, iUnit, ISH FW, etc. This structure will appear after manifest header for codepartitions as shown in Figure 18.

**Table 4: Signed Package Info Extension**

Name	Offset (Dec)	Offset (Hex)	Size (bytes)	Description
Extension Type	0	0	4	= 15 for Signed Pkg Info Extension
Extension Length	4	4	4	In bytes; equals (52 + 52*n) for this version, where 'n' is the number of modules in the manifest



Name	Offset (Dec)	Offset (Hex)	Size (bytes)	Description
Package Name	8	8	4	Name of the package
Version Control Number (VCN)	12	C	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	10	16	Bitmap of usages depicted by this manifest, indicating which key is used to sign the manifest
SVN	32	20	4	SVN of this signed image
Reserved	36	24	16	Must be 0
Module 0 Name	52	34	12	Character array; if name length is shorter than field size, the name is padded with 0 bytes.
Module 0 Type	64	40	1	0 – Process 1 – Shared Library 2 – Data 3 – Reserved...
Module 0 Hash Algorithm	65	41	1	2 = SHA256
Module 0 Hash Size	66	42	2	Size of Hash in bytes = N. N = 32
Module 0 Metadata Size	68	44	4	Size of metadata file
Module 0 Metadata Hash	72	48	32	The SHA2 of the module metadata file
...				

### 9.4.3 OEM Key Manifest

After Manifest Header for OEM KM, there will be Key Manifest Extension that is used for OEM KM as shown in Figure 17.

**Table 5: Key Manifest Extension**

Name	Offset (Dec)	Offset (Hex)	Size (bytes)	Description
Extension Type	0	0	4	= 14 for Key Manifest Extension
Extension Length	4	4	4	In bytes; equals $(36 + 68*n)$ for this version, where 'n' is the number of keys in the OEM KM manifest
Key Manifest Type	8	8	4	2 = OEM Key Manifest
Key Manifest Security Version Number (KMSVN)	12	C	4	The security version number for the OEM Key Manifest
Reserved	16	10	2	0 – Reserved





Name	Offset (Dec)	Offset (Hex)	Size (bytes)	Description
Key Manifest ID	18	12	1	ID number of the Key Manifest. This is matched by the verifier against the value stored in the platform in FPF. This is typically used as an ODM ID – to enable an OEM to assign IDs to its various ODMs and generate Key Manifests specific to each ODM.
Reserved	19	13	1	Must be 0
Reserved	20	14	16	Must be 0
Key 0 Usage	36	24	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) Bit 42: ISH manifest 2 (ISH BUP) Bit 43: OEM Debug Tokens Manifest Bit 44: Reserved Bit 45: Reserved Bit 46: Reserved Bit 47: OEM Key Attestation Bit 48: OEM DAL Manifest Bit 49 - 127: Reserved for future OEM usages
Key 0 Reserved	52	34	16	
Key 0 Reserved	68	44	1	
Key 0 Hash Algorithm	69	45	1	2 = SHA256
Key 0 Hash Size	70	46	2	Size of Hash in bytes = N. N = 32
Key 0 Hash	72	48	N (32)	The hash of the key.
...				

## 9.5 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>
```





## 10 Common Bring Up Issues and Troubleshooting Table

---

### 10.1 Common Bring Up Issues and Troubleshooting Table

Problem / Issue	Solution / Workaround
Intel MEU tool fails to run	Confirm that the MEU_Config and template xml files are present in the same folder as the Intel MEU tool. Confirm that both files have been modified properly.

§