

Apollo Lake Signing and Manifesting Guide

User Guide

Revision 1.1

September 2016

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 © 2016, Intel Corporation. All rights reserved.



Contents

1	Introduction	6
1.1	Goal	6
1.2	Pre-Requisites	6
1.3	Tools Used In This Document	6
1.4	Terminology	7
2	Manifesting and Signing OEM Components in the IFWI Image	8
2.1	Creating a Signed IFWI Image.....	8
2.2	Creating an Unsigned IFWI Image.....	9
3	Creating PKI Key Pairs.....	11
3.1	Introduction	11
3.2	Generating Key Pair for Signing	11
3.3	Creating the Public Key Hash:	11
3.3.1	Creating Public Key Hash Using Intel® MEU	11
3.3.2	Creating Public Key Hash Manually.....	12
3.4	Key Security.....	13
4	Using Intel® MEU to Create and Add Manifests to Binaries and Sign Binaries	14
4.1	Introduction	14
4.2	Binary Manifesting Signing Overview	14
4.3	Intel® MEU Configuration	15
4.4	Intel® MEU Usage Flow	16
4.5	Intel® MEU Decomposition	17
4.6	Intel® MEU Resign.....	17
4.6.1	Secure Signing for SMIP	18
4.7	Different Binary Types Supported By Intel® MEU.....	19
4.7.1	IAFW/BIOS	19
4.7.2	ISH.....	22
4.7.3	IUnit / aDSP.....	23
4.7.4	SMIP	24
5	OEM Key Manifest.....	26
5.1	Introduction	26
5.2	Creation of Manifest	26
6	Add Components to Intel® FIT.....	30
6.1	Introduction	30
6.2	Include each Binary Component.....	30
6.3	Add the <i>Private</i> Key for SMIP.....	30
6.4	Add the OEM Key Manifest	31
6.5	Add the Public Key Hash for OEM Key Manifest	31
6.6	Change the Key Manifest ID	32
6.7	Enable Boot Guard	32
6.8	Configure Intel FIT to call Intel® MEU to Sign and Manifest the SMIP	32
6.9	Add Debug Token	33
7	Creation of Update Image.....	34
7.1	Introduction	34
7.2	DnX	34



	7.2.1	DnX Image Creation Using Intel® MEU	34
	7.2.2	DnX Image Creation Using Intel FIT	35
	7.3	BIOS Capsule Update	36
8		Using Intel MEU with Other Signing Tools	37
	8.1	Introduction	37
	8.2	Creating Keys and Hashes.....	37
	8.3	Create Manifested Binaries.....	37
	8.4	Export Manifests	37
	8.5	Sign Manifests	38
	8.6	Import Manifest	39
9		Common Bring Up Issues and Troubleshooting Table.....	40
	9.1	Common Bring Up Issues and Troubleshooting Table.....	40

Figures

Figure 1.	High Level Overview of Manifesting and Signing OEM Components in the IFWI Image.....	9
Figure 2.	Schematic View of Manifesting and Signing Process	15
Figure 3.	Intel MEU Configuration xml	16
Figure 4.	Intel MEU list of Supported Binary Types.....	17
Figure 5.	Default BIOS xml	20
Figure 6.	BIOS xml Edited to Accept all Possible Binary Components	21
Figure 7.	Code Partition xml	22
Figure 8.	Code Partition Metadata xml	24
Figure 9.	Default OEM Key Manifest XML.....	27
Figure 10.	OEM Key Manifest with 3 Different Signing Keys	28
Figure 11.	Intel FIT fields to enter IUnit, PMC and uCode Binaries	30
Figure 12.	Entering SMIP private key	31
Figure 13.	Entering OEM Key Manifest	31
Figure 14.	Entering OEM Public Key Hash.....	32
Figure 15.	Entering OEM Public Key Hash.....	32
Figure 16.	Configuring Intel FIT to sign the SMIP	33
Figure 17.	Adding a Debug Token	33
Figure 18.	DnX xml	35

Tables

Table 1.	Components Recognized by Intel MEU, and How Key Manifests should Handle	28
----------	---	----



Revision History

Revision Number	Description	Revision Date
0.1	Initial Release	August 2015
0.2	Completely revised, with screen shots for and steps for each file type signed by MEU, details on creating keys and hashes, and explanations of all the hashes included in the OEM Key Manifest Also added information of the token fields.	August 2015
0.3	Removed use of MEU to create tokens	September 2015
0.7	Updated list of binary hashes to be included in OEM Key Manifest Updated FIT screenshots	November 2015
0.8	Updated MEU support for hash creation, decomposition, export/import, resigning, clarifications on BIOS BPM	November 2015
0.9	Minor updates for APL Beta release	January 2016
0.95	Signing made optional. OEM Key Manifest does not include binaries provided by Intel and not changed by OEM	March 2016
1.0	Rev up for PV to 1.0	June 2016
1.1	Added secure signing flow in section 2.6.1	September



1 Introduction

This document gives an overview of the process of manifesting and signing OEM components that then will be included in the IFWI image for Apollo Lake platforms.

OEMs are always required to add manifests to components in the IFWI images. However, they are not required to sign components, or add an OEM Key Manifest, unless they wish to

- enable Secure Boot
- make use of OEM Debug Unlock tokens
- replace any of the Intel provided binaries (such as iUnit, aDSP, ISH)

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

1.1 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 IFWI image

1.2 Pre-Requisites

The user should download and install the following kit.

- Latest Intel® TXE FW kit

The kit can be downloaded from the following location:

<https://platformsw.intel.com/>

The overall platform bring-up procedure is described in:

- APL Firmware Bring Up Guide

The System Tools User Guide gives further detail on the usage of all firmware manufacturing and is the definitive guide to the details of each tool's usage:

- APL System Tools User Guide

1.3 Tools Used In This Document

The following tools are used within this document:

- Intel® Flash Image Tool (Intel® FIT) [Intel® TXE Kit]
- Intel® Manifest Extension Utility (Intel® MEU) [Intel® TXE Kit]
- Open SSL [Open Source]



1.4 Terminology

Term	Description
Intel® FIT	Intel® Flash Image Tool
IBB	Initial Boot Block
IBBL	Initial Boot Block Loader
IFWI	Integrated Firmware Image
ISH	Integrated Sensor Hub
OBB	
Intel® MEU	Intel® Manifest Extension Utility
SUT	System Under Test

§



2 Manifesting and Signing OEM Components in the IFWI Image

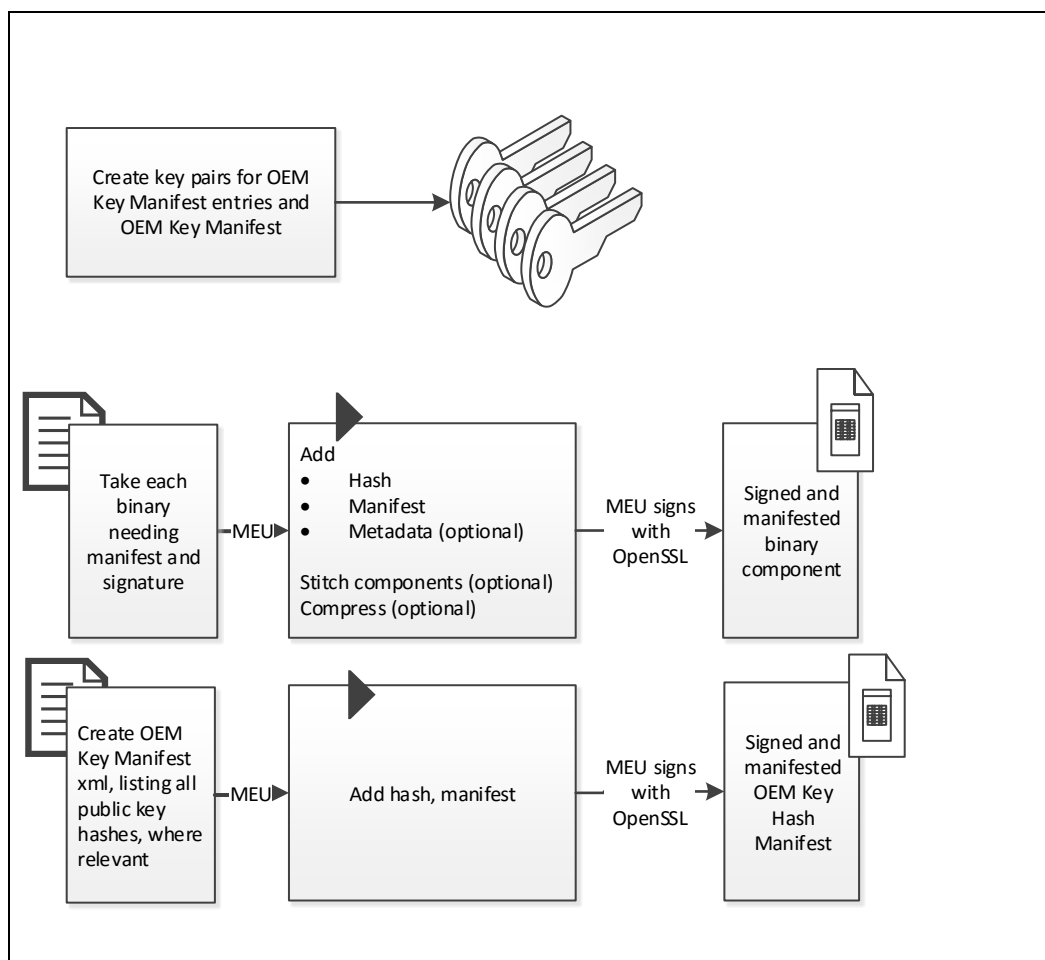
2.1 Creating a Signed IFWI Image

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 5.2.
 - 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, including within it the public key hash of each of the created keys, and use the Intel MEU to sign it. Note: The order in which steps 2 and 3 are executed does not matter.
4. Add each binary component to the Intel FIT
5. Add to Intel FIT the **private** key to sign the SMIP. This is private key of one of the key pairs already created in step 1.a and whose public key hash is included in the OEM Key Manifest in step 3.
6. Add to Intel FIT the OEM Key Manifest created in step 3.
7. Add to Intel FIT the public key hash for the OEM Key Manifest. This will be burned into an IFP when the system closes manufacture, and can never be changed after this stage.
8. Configure Intel FIT to be able to call the Intel MEU to sign and manifest the SMIP.
9. In some cases, add a debug token to Intel FIT, to allow the image to be debugged. Note that in general, debug tokens can be injected into the system post-manufacture, as needed, and not be included in the Intel FIT.
10. If using Intel FIT to create a DnX image, configure Intel FIT to also build, manifest and sign such an image during compilation time. This includes supplying Intel FIT with the **private** key for signing the DnX image, which is the same key used for signing the OEM Key Manifest.



Figure 1. High Level Overview of Manifesting and Signing OEM Components in the IFWI Image



2.2 Creating an Unsigned IFWI Image

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

1. Use the Intel MEU tool to add to each binary a manifest, and where relevant also add metadata, stitch and/or compress the binary.
2. Add each binary component to the Intel FIT
3. If using Intel FIT to create a DnX image, configure Intel FIT to also build, manifest and sign such an image during compilation time.

Creating an unsigned IFWI image skips the following steps required in the creation of a signed IFWI image:

- Generation of PKI key pairs and their public key hashes.
- Using the Intel MEU tool to add to each binary a signed manifest. You still need to use the Intel MEU tool to add to each binary a manifest, and where relevant also add metadata, stitch and/or compress the binary.



Manifesting and Signing OEM Components in the IFWI Image

- Creation of an OEM Key Manifest.
- Adding to Intel FIT the private key to sign the SMIP.
- Adding to Intel FIT the OEM Key Manifest.
- Adding to Intel FIT the public key hash for the OEM Key Manifest.
- Configuring Intel FIT to be able to call the Intel MEU to sign the SMIP. However, Intel FIT must still be configured to be able to call the Intel MEU to manifest the SMIP.
- Supplying Intel FIT with the **private** key for signing a DnX image, if the target platform does not expect signed images.

§



3 Creating PKI Key Pairs

3.1 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. These are enumerated in Section 5.2.
2. The OEM Key Manifest

3.2 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, and it must be installed separately. One source for 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`

3.3 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.

3.3.1 Creating Public Key Hash Using Intel® MEU

You can create the public key hash using 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: 3.0.0.1048
Copyright (c) 2013 - 2015, Intel Corporation. All rights reserved.
10/29/2015 - 10:10:24 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.
-----
```

3.3.2 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.

3.4 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 be always 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.

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, despite Intel FIT and Intel MEU needing access to them while signing components and building the image. For example, the Intel FIT and MEU could be run on a secure server which houses the keys.

OEMs should use a set of keys during the development process, and then a separate set of keys for creating production images. This will ensure that on production platforms, only the production OEM Key Manifest, with signatures for production components, can be run.

§



4 *Using Intel® MEU to Create and Add Manifests to Binaries and Sign Binaries*

4.1 Introduction

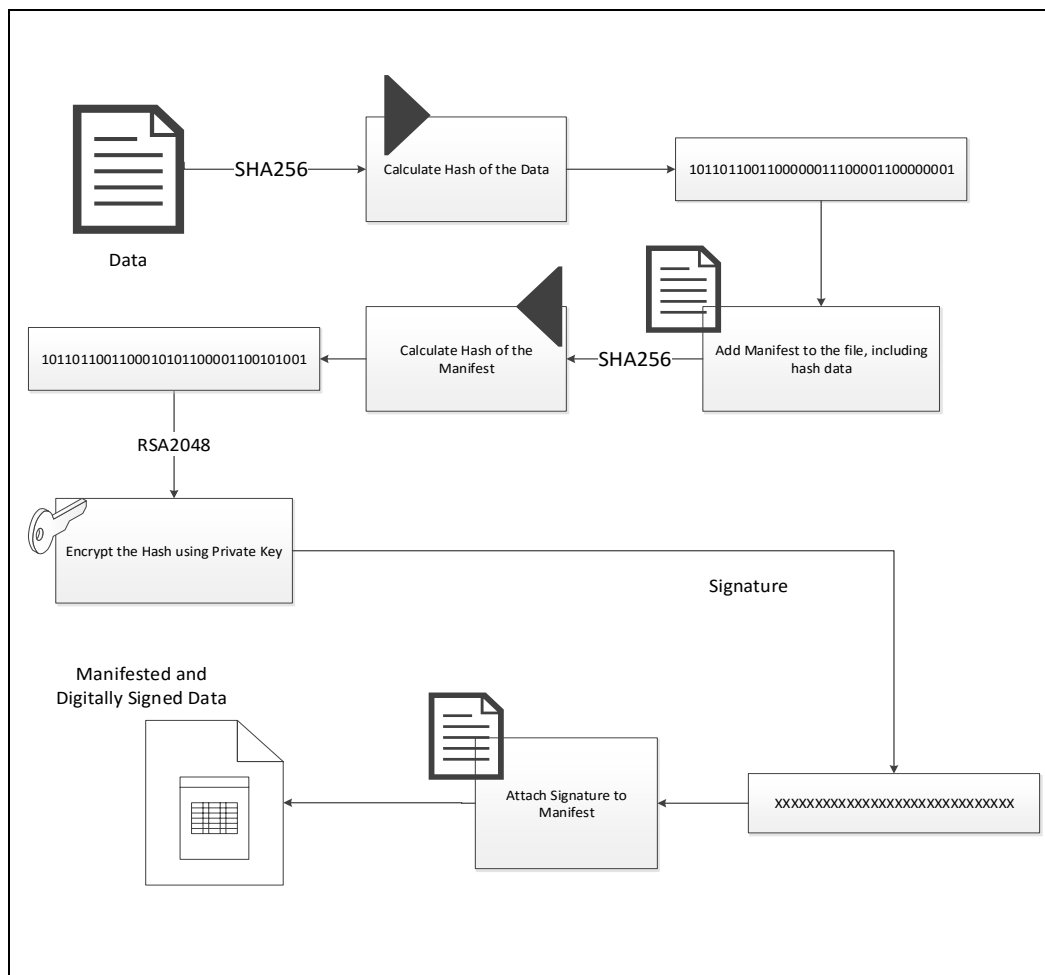
All of the OEM Key Manifest components owned by an OEM are expected to have a manifest added. If the IFWI image is signed, all of the components also need to be signed. Intel provides the Intel MEU to assist with this task. The Intel MEU is also able to add required manifests, metadata (where relevant), as well as compress and stitch binary components (where relevant).

4.2 Binary Manifesting Signing Overview

Intel signing for APL platforms employs 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 **Error! Reference source not found.** below. The Intel MEU is used to create the manifests, and interfaces with OpenSSL to add signatures to the manifest.



Figure 2. Schematic View of Manifesting and Signing Process



4.3 Intel® MEU Configuration

To use Intel MEU, you first need to configure the tool. To do this, run the following command:

```
# meu -gen meu_config
```

This will generate a default configuration xml file:



Figure 3. Intel MEU Configuration xml

```
<?xml version="1.0" encoding="UTF-8"?>
- <MeuConfig version="2.6">
  - <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="Path to signing tool executable."
      value="C:\openssl\openssl.exe" />
    <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="C:\keys\priv_key.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 edit the 'SigningTool' node to be '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.

4.4 Intel® MEU Usage Flow

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 4. Intel MEU list of Supported Binary Types

```

C:\WINDOWS\system32\cmd.exe

D:\Engineering\MEU>meu -binlist
=====
Intel(R) Manifest Extension Utility. Version: 3.0.0.1044
Copyright (c) 2013 - 2015, Intel Corporation. All rights reserved.
10/14/2015 - 10:43:01 am
=====

Command Line: meu -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)
  Bios                - Single BIOS binary for use with FII
  CodePartition        - Generic Updateable Code Partition
  CodePartitionMeta    - Updateable Code Partition with user-provided Metadata
  DeviceLifecycleToken - Device Lifecycle Token
  DnxRecoveryImage     - DNX Recovery IFWI Image
  OEMKeyManifest       - OEM Key Manifest
  OemUnlockToken       - OEM Unlock Token

Program terminated.

D:\Engineering\MEU>

```

For each file that needs to be manifested and signed, you use Intel MEU to generate an xml for that file type, and then edit the xml to ensure the data is correct – in particular that it includes the path to the relevant file. You then call MEU with the edited xml as input, and pass in the name of the required output file, and 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, but override the private key used for signing on the command line:

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

4.5 Intel® MEU Decomposition

Intel MEU is able to decompose a manifested and signed binary, to return it to the original state it was in before Intel MEU added a manifest and/or signature, together with an xml detailing the decomposition. This xml can later be used as input to 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 a BIOS binary, you can call:

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

4.6 Intel® MEU Resign

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



```
# 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 to 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 resigned, or `'all'` if all are to be resigned (using the new key). If the index, or `'all'` is not included, 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	0x000002058	0x000000378	SMIP.man
1	0x000006058	0x000000378	RBEP.man
2	0x00000E088	0x0000003E0	PMCP.man
3	0x00001C130	0x000000D6C	FTPR.man
4	0x00006F000	0x0000002EC	rot.key
5	0x000072CD0	0x0000003B8	oem.key
6	0x000077070	0x0000002EC	IBBP.man
7	0x0000D1058	0x000000378	ISHC.man
8	0x0001116E8	0x0000011B0	NFTP.man
9	0x0005C2070	0x000000378	IUNP.man

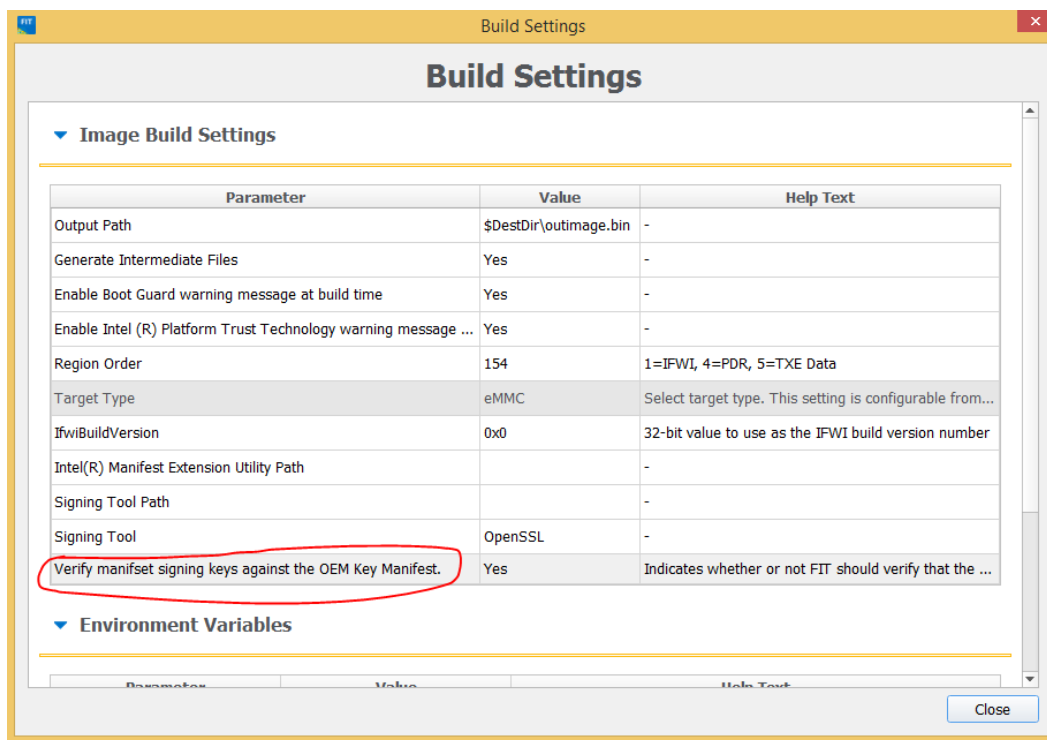
The Intel® MEU can then be called again, including the index desired. Following the above example, if the SMIP is to be resigned, call:

```
# meu.exe -resign 0 -f <input.bin> -o <output.bin> -key <privatekey.pem>
```

4.6.1 Secure Signing for SMIP

FIT requires SMIP private key to be provided in order to build SMIP partition. If FIT is not running in secure environment, you may exercise secure signing by giving a dummy private (non-production) key to FIT different than your production key (as the one in OEM KM). FIT validates all the manifests in the image by default before building the image. So in order to allow FIT to continue building the image without erroring out (due to manifest mismatch), this can be done by disabling the option in FIT "Build Settings" setting "Verify manifest signing keys against the OEM Key Manifest" to "No".

This allows you to resign SMIP partition in your production IFWI using the SMIP production key after the image has been created by FIT.



So to resign SMIP in production IFWI with production key, you may call:

```
# meu.exe -resign 0 -f <input.bin> -o <output.bin> -key <privatekey.pem>
```

Note: This disables FIT feature of validating IFWI manifests and key mismatching.

4.7 Different Binary Types Supported By Intel® MEU

Intel MEU is able to add manifests and sign several types of files, as enumerated below. There are also other binaries that may be signed by other tools (such as BIOS Capsule creation tools, OS loader creation tools etc.), but which Intel MEU does not sign.

Note: Some firmware image binary components can be created by Intel. In all cases of binaries provided by Intel, the binary will already have a manifest and signature, and OEMs do not need any further processing on these binaries. The hashes for any binary file provided by Intel that can be replaced by an OEM binary should be included in the OEM Key Manifest (see explanation below on the OEM Key Manifest). This applies to all binaries provided by Intel except for CSE and PMC binaries, whose hashes are handled internally via Intel Key Manifests, and should not be included in the OEM Key Manifest.

4.7.1 IAFW/BIOS

In APL platforms, IAFW/BIOS is composed of IBB and OBB sub-partitions. The IBB sub-partitions includes a Boot Policy Manifest (BPM.met.bin) module, as well as IBBL and IBB data modules. The OBB sub-partition can be a single module, or itself



In order to support Boot Guard, the BPM.met needs to contain hashes of the IBB, IBBL and OBB partitions in the BIOS image. The creation of BPM.met, and the calculation and population of the hash fields is not done using Intel® MEU. Please contact your BIOS vendor or AE for details on how to create the BPM.met module.

Intel MEU is able to stitch the BIOS components into a single binary. It creates the manifest file for the BIOS, signs and stitches together all of the components.

```
# meu -gen Bios
```

```
<?xml version="1.0" encoding="UTF-8"?>
- <Bios version="2.6">
  - <IbbSubPartition label="IBB">
    <Length help_text="Set the length of sub partition." value="0x0"/>
    <Usage value="BootPolicyManifest"
      value_list="CseBspManifest,,CseMainManifest,,PmcManifest,,BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMainFwManifest,,c
    <VendorId value="0x8086"/>
    <SecurityVersionNumber label="Secure Version Number" value="0"/>
    <VersionControlNumber label="Version Control Number" value="0"/>
    <VersionMajor label="Version Major" value="0"/>
    <VersionMinor label="Version Minor" value="0"/>
    <VersionHotfix label="Version Hotfix" value="0"/>
    <VersionBuild label="Version Build" value="0"/>
  - <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>
  - <BootPolicyManifest>
    <Enabled help_text="If set to 'Disabled' the Boot Policy Manifest will not be created and thus the IBB, IBBL and OBB modules
      will not be covered by the manifest signature." value="true" value_list="true,,false"/>
  </BootPolicyManifest>
  - <Modules>
    - <DataModule name="IBBL">
      <InputFile value="IBBL.bin"/>
    </DataModule>
  </Modules>
</IbbSubPartition>
- <ObbSubPartition label="OBB">
  <Length help_text="Set the length of sub partition." value="0x0"/>
  - <Modules>
    - <DataModule name="OBB">
      <InputFile value="OBB.bin"/>
    </DataModule>
  </Modules>
</ObbSubPartition>
</Bios>
```

The BIOS.xml file generated can then be edited to ensure the input files are correct.

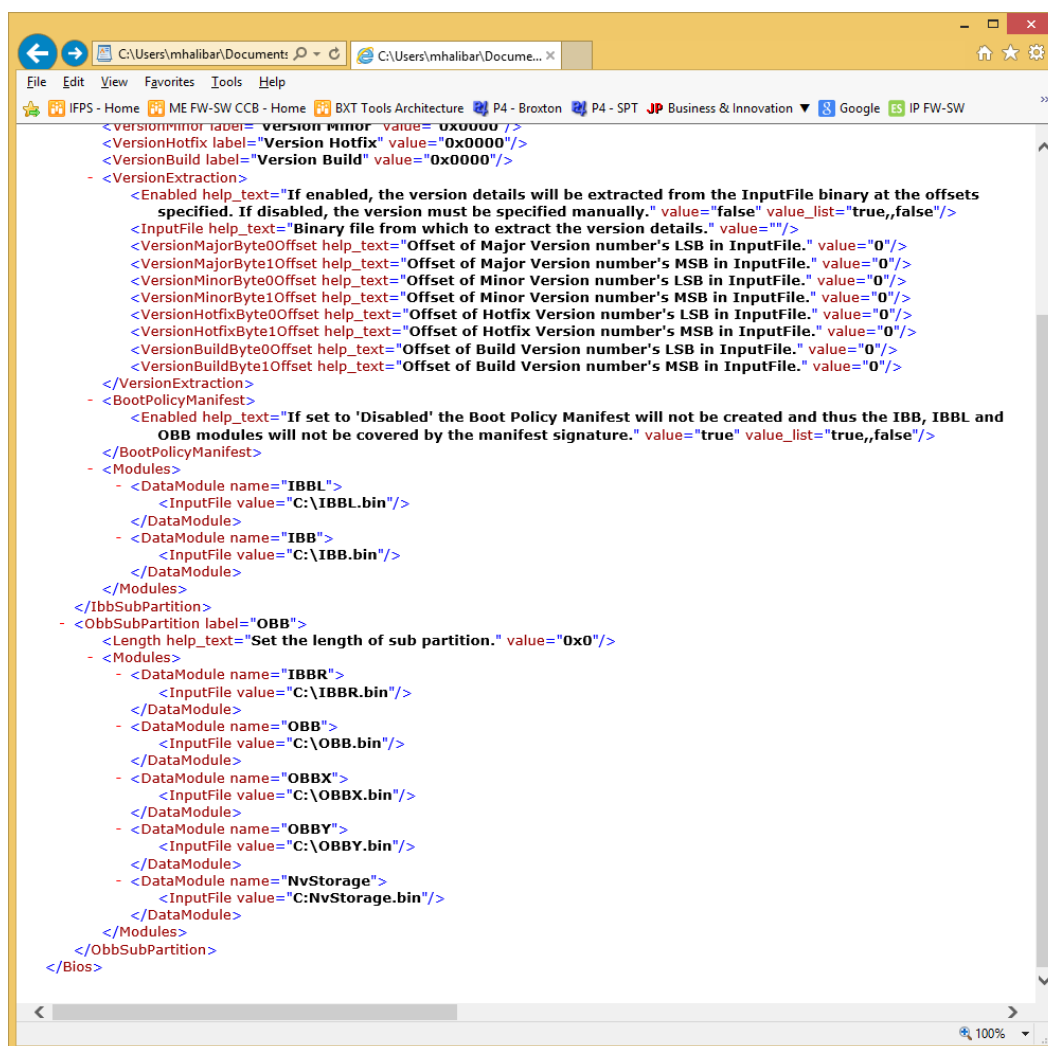


The default BIOS.xml file assumes that the only modules to be stitched are IBBL.bin and OBB.bin. It does not include fields for all possible sub-modules, as the number and identity of such binaries depends on how the BIOS is built.

For example, the default template has a single DataModule node under the IBB Sub Partition node. If the BIOS has the IBB and IBBL as separate binaries, an extra node will need to be added.

Likewise, under the OBB sub partition node, the default template has a single DataModule node (IBBL), while the BIOS may need entries for IBBR, OBB, OBBX, OBBY and NvStorage. If the BIOS is to be created by stitching together these extra binaries, they can be added to the xml file.

Figure 6. BIOS xml Edited to Accept all Possible Binary Components



The xml should also be edited to indicate if a Boot Policy Manifest (BPM) should be created and stitched into the BIOS binary. A BPM is required if Boot Guard will be enabled on the platform, otherwise it does not need to be created.



4.7.1.1.1 Example of node for creation of BPM

```
<BootPolicyManifest>
  <Enabled value=" true" value_list="true,,false"
help_text="If set to 'Disabled' the Boot Policy Manifest will not be
created and thus the IBB, IBBL and OBB modules will not be covered by
the manifest signature." />
</BootPolicyManifest>
```

4.7.1.1.2 Example of node for non-creation of BPM

```
<BootPolicyManifest>
  <Enabled value="false" value_list="true,,false"
help_text="If set to 'Disabled' the Boot Policy Manifest will not be
created and thus the IBB, IBBL and OBB modules will not be covered by
the manifest signature." />
</BootPolicyManifest>
```

Once the BIOS xml has been edited to include all the required input files, and create the BPM if desired, 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 <BIOS.xml> -o <BIOS.bin> -key<privatekey.pem>
```

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

4.7.1.2 Using Intel® MEU to Sign an already-stitched BIOS Binary

Intel® MEU is able to sign a full BIOS binary that has already been stitched together. For example, an IBV may deliver a fully stitched binary to an OEM, who will then want to sign it. The signing functionality is not dependent on whether the BIOS has already been signed or not – in all cases, a new signature is placed in the manifest of the binary.

To sign, or resign a binary, follow the instructions in section 4.6.

4.7.2 ISH

The ISH binary is regarded as a 'code partition' by Intel MEU, and an xml template can be generated for it 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, the path to the LZMA compression file also needs to be entered. Note that Intel MEU tool only supports the LZMA tool, provided by Intel, to compress binaries. The ISH binary requires compression.

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,,BootPolicyManifest,,iUnitBootLoaderManifest,,iUnitMain
  <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="true,false"/>
    <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 enabled="true" 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 it 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 to that defined in the default Intel MEU configuration xml.

4.7.3 IUnit / aDSP

The IUnit and aDSP binaries are regarded as a 'code partition metadata' by Intel MEU, and an xml template can be generated for it using the following command

```
# meu -gen CodePartitionMeta
```




the path to the Intel MEU, which in turn passes it to the signing tool. The path to the key is entered in the Platform Protection tab in Intel FIT.





5 OEM Key Manifest

5.1 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 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 save and protect 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 DnX, or other flash burning mechanism), it will need to be signed with this key.

5.2 Creation of Manifest

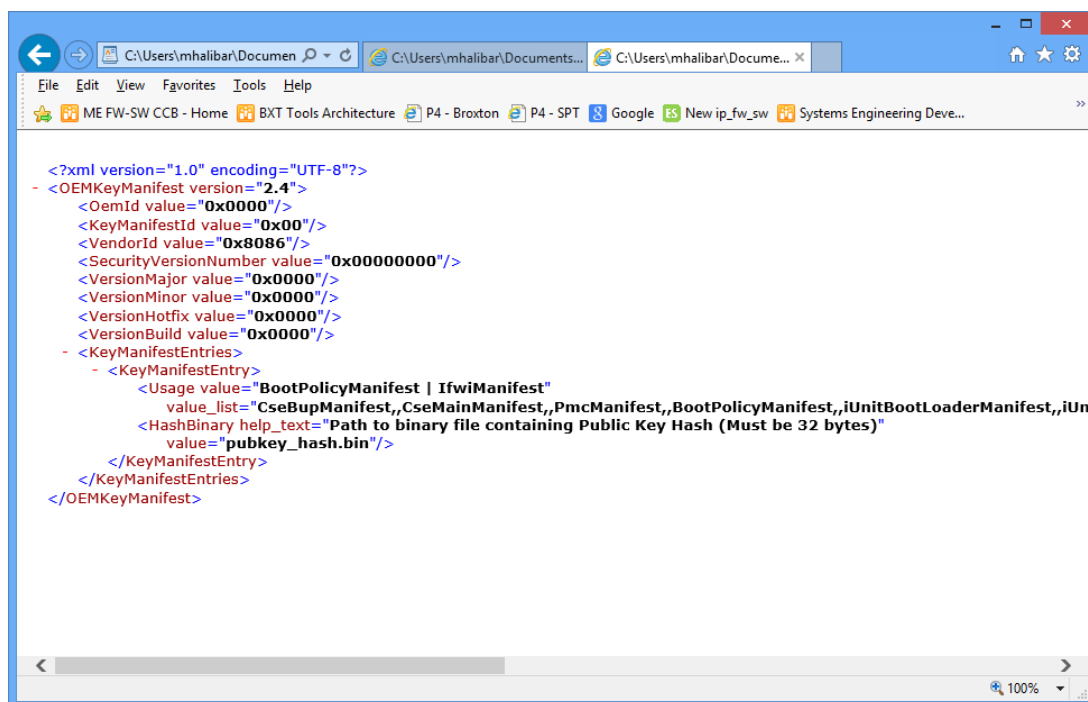
The manifest file xml template can be generated using the following command:

```
# 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 9. Default OEM Key Manifest XML



The KeyManifestId field must not be left with its default value of 0x0, and 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 for which there is 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, where BootPolicyManifest and IfwiManifest both share the same pubkey_hash.bin.

So, for example, if the OEM Key Manifest wants to have

- BootPolicyManifest and IFWIManifest signed with key 1
- IshManifest and ISHBupManifest signed with key 2
- OemSmipManifest signed with key 3

It will appear as follows:



Figure 10. OEM Key Manifest with 3 Different Signing Keys

```
<?xml version="1.0" encoding="UTF-8"?>
- <OEMKeyManifest version="2.5">
  <OemId value="0x0000"/>
  <KeyManifestId value="0x00"/>
  <VendorId value="0x8086"/>
  <SecurityVersionNumber value="0x00000000"/>
  <VersionMajor value="0x0000"/>
  <VersionMinor value="0x0000"/>
  <VersionHotfix value="0x0000"/>
  <VersionBuild value="0x0000"/>
  - <KeyManifestEntries label="KeyManifestEntries">
    - <KeyManifestEntry label="KeyManifestEntry">
      <Usage value="BootPolicyManifest | IfwiManifest"
        value_list="CseBupManifest,,CseMainManifest,,PmcManifest,,BootPolicyManifest,,iUnitBootLoaderManifest"
        <HashBinary value="pubkey_hash1.bin" help_text="Path to binary file containing Public Key Hash (Must be 32 bytes)"/>
      </KeyManifestEntry>
    - <KeyManifestEntry label="KeyManifestEntry">
      <Usage value="IshManifest | IshBupManifest"
        value_list="CseBupManifest,,CseMainManifest,,PmcManifest,,BootPolicyManifest,,iUnitBootLoaderManifest"
        <HashBinary value="pubkey_hash2.bin" help_text="Path to binary file containing Public Key Hash (Must be 32 bytes)"/>
      </KeyManifestEntry>
    - <KeyManifestEntry label="KeyManifestEntry">
      <Usage value="OemSmipManifest"
        value_list="CseBupManifest,,CseMainManifest,,PmcManifest,,BootPolicyManifest,,iUnitBootLoaderManifest"
        <HashBinary value="pubkey_hash3.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, and for which key hashes can be entered are:

Table 1. Components Recognized by Intel MEU, and How Key Manifests should Handle

Entry Name	Meaning	Who Creates	Include in OEM Key Manifest?
BootPolicyManifest	IFWI/BIOS	OEM	Yes
iUnitBootLoaderManifest	Camera firmware boot loader	Intel if binary provided by Intel, OEM if OEM using a proprietary binary	Only if OEM is replacing this binary with his own version
iUnitMainFwManifest	Camera main firmware	Intel if binary provided by Intel, OEM if OEM using a proprietary binary	Only if OEM is replacing this binary with his own version



Entry Name	Meaning	Who Creates	Include in OEM Key Manifest?
cAvsImage0Manifest	Audio (aDSP) firmware 0	Intel if binary provided by Intel, OEM if OEM using a proprietary binary	Only if OEM is replacing this binary with his own version
cAvsImage1Manifest	Audio (aDSP) firmware 1	Intel if binary provided by Intel, OEM if OEM using a proprietary binary	Only if OEM is replacing this binary with his own version
IfwiManifest	For the creation of an update image for use via BIOS capsule update	OEM	Yes
OsBootLoaderManifest	OS Boot loader	OEM	Yes
OsKernelManifest	OS Kernel	OEM	Yes
OemSmipManifest	SMIP includes many of the settings defined in Intel FIT	OEM	Yes
IshBupManifest	Integrated Sensor Hub bring up firmware.	Intel	No
OEMDebugManifest	OEM Debug token	OEM	Yes

In APL, binary files provided by Intel do not need to have their hashes included in the OEM Key Manifest, unless replaced by the OEM, as called out in the table above.

Not every hash listed in the table above needs to be entered – 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 load, if attempted. Likewise, if the OEM is not using Intel APIs to verify 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 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 it with the private key created for this purpose.

```
# 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 to that defined in the default Intel MEU configuration xml.



6 Add Components to Intel® FIT

6.1 Introduction

Intel FIT is a tool provided to OEMs to stitch together multiple binary files, configuration data and other input into a full IFWI image. Not every image will have every component e.g. only images including Intel ISH need to include an ISH image binary in Intel FIT. This document will only discuss the usage of the tool as relevant to the signing mechanism. The full image creation procedure is detailed in the Apollo Lake - Intel® Trusted Execution Engine (Intel® TXE) Firmware Bring-Up Guide.

6.2 Include each Binary Component

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

Figure 11. Intel FIT fields to enter IUnit, PMC and uCode Binaries

The screenshot shows a web interface with five expandable sections, each containing a table with three columns: Parameter, Value, and Help Text.

- SMIP Sub-Partition**

Parameter	Value	Help Text
IAFW SMIP Binary File		-
PMC SMIP Binary File		-
ME GPIO SMIP Binary File		-
- IUnit Sub-Partition**

Parameter	Value	Help Text
IUnit Binary File		-
- PMC Sub-Partition**

Parameter	Value	Help Text
PMC Binary File		-
- uCode Sub-Partition**

Parameter	Value	Help Text
uCode Patch 1 Input File		-
uCode Patch 2 Input File		-
- Intel(R) TXE Sub-Partition**

6.3 Add the *Private* Key for SMIP

Add to Intel FIT the private key for SMIP. The field is available in the Platform Protection tab

Since Intel FIT creates the SMIP binary, and calls Intel MEU to add a manifest and signature to it, it needs access to the private key for signing SMIP.



Figure 12. Entering SMIP private key

Note: If the image will not be signed, this field should be left empty.

▼ Platform Integrity

Parameter	Value	Help Text
SMIP Signing Key		-
OEM Public Key Hash	00 00 00 00 00 00 00 00 00 ...	This option is for entering the raw hash string or certificate file for the SHA-256 hash of the OEM ...
Oem Key Manifest Binary		-

6.4 Add the OEM Key Manifest

This hash will be burned into an IFP when the system closes manufacture, and can never be changed after this stage.

Note: If the image will not be signed, this field should be left empty.

Figure 13. Entering OEM Key Manifest

▼ Platform Integrity

Parameter	Value	Help Text
SMIP Signing Key		-
OEM Public Key Hash	00 00 00 00 00 00 00 00 00 ...	This option is for entering the raw hash string or certificate file for the SHA-256 hash of the OEM ...
Oem Key Manifest Binary		-

6.5 Add the Public Key Hash for OEM Key Manifest

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

This hash will be burned into an IFP when the system closes manufacture, and can never be changed after this stage.

Note: If the image will not be signed, this field should be left empty.



Figure 14. Entering OEM Public Key Hash

▼ Platform Integrity

Parameter	Value	Help Text
SMIP Signing Key		-
OEM Public Key Hash	00 00 00 00 00 00 00 00 ...	This option is for entering the raw hash string or certificate file for the SHA-256 hash of the OEM ...
OEM Key Manifest Binary		-

6.6 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.

Note: If the image will not be signed, this field should be set to 0.

Figure 15. Entering OEM Public Key Hash

▼ Boot Guard Configuration

Parameter	Value	
Key Manifest ID	0x0	This option is for enter

6.7 Enable Boot Guard

Boot Guard can be enabled using the 'Boot Guard Profile Configuration' field in the Platform Protection tab. Legacy mode essentially disables Boot Guard, and the components within the BIOS image are then not verified before loading. If Boot Guard is enabled, each component within the BIOS is validated before loading.

Note: If the image will not be signed, Boot Guard cannot be enabled

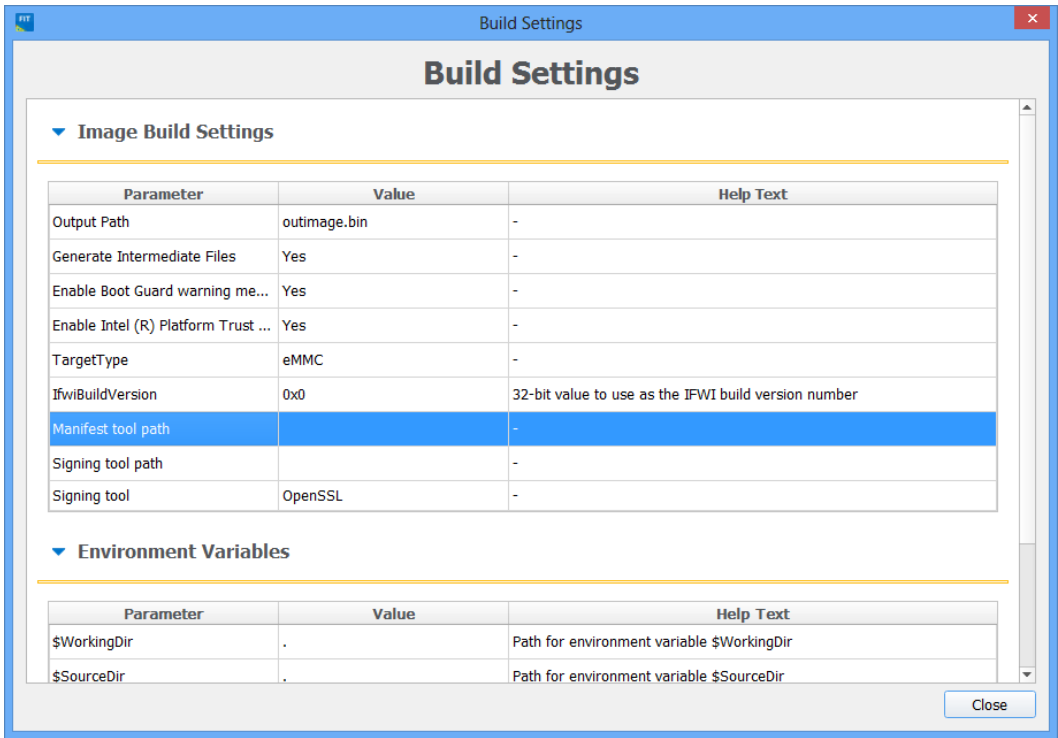
6.8 Configure Intel FIT to call Intel® MEU to Sign and Manifest the SMIP

Configure Intel FIT to be able to call the Intel MEU to sign and manifest the SMIP. This is done via the Build Settings dialog. You need to enter the path to the Intel MEU, the path to the signing tool, and the identity of the signing tool (currently only OpenSSL is supported).

Note: If the image will not be signed, the signing tool fields should be left empty. The manifest tool path still needs to be configured.



Figure 16. Configuring Intel FIT to sign the SMIP

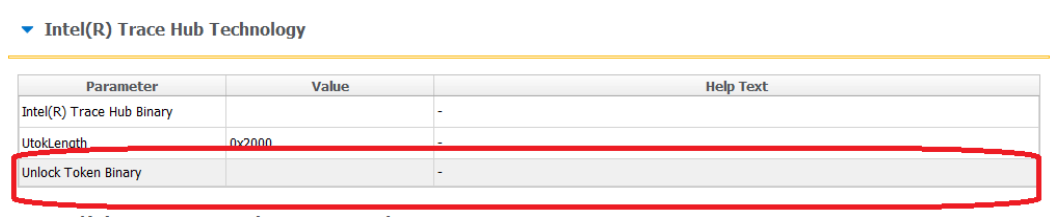


6.9 Add Debug Token

In some cases, add a debug token to Intel FIT, to allow the image to be debugged. This field is available in the Debug tab. Note that in general, debug tokens will be injected into the system post-manufacture, as needed.

Note: If the image will not be signed, this field cannot be used.

Figure 17. Adding a Debug Token





7 Creation of Update Image

7.1 Introduction

Apollo Lake platforms accept image updates via 2 protocols

- DnX
- BIOS Capsule

7.2 DnX

Apollo Lake platforms, using UFS or eMMC flash devices, support burning of IFWI images using the DnX protocol. The IFWI image supplied must be manifested, and include in its manifest an OEM ID and platform ID. If the target platform had a signed image on it, with the FPF burned, the DnX image must also be signed. These values are checked by the DnX protocol against data burned into FPF fuses, and only if they match will the image burn be accepted.

Important

Ensure that the key used for signing the DnX image is the same key used for signing the OEM Key Manifest.

7.2.1 DnX Image Creation Using Intel® MEU

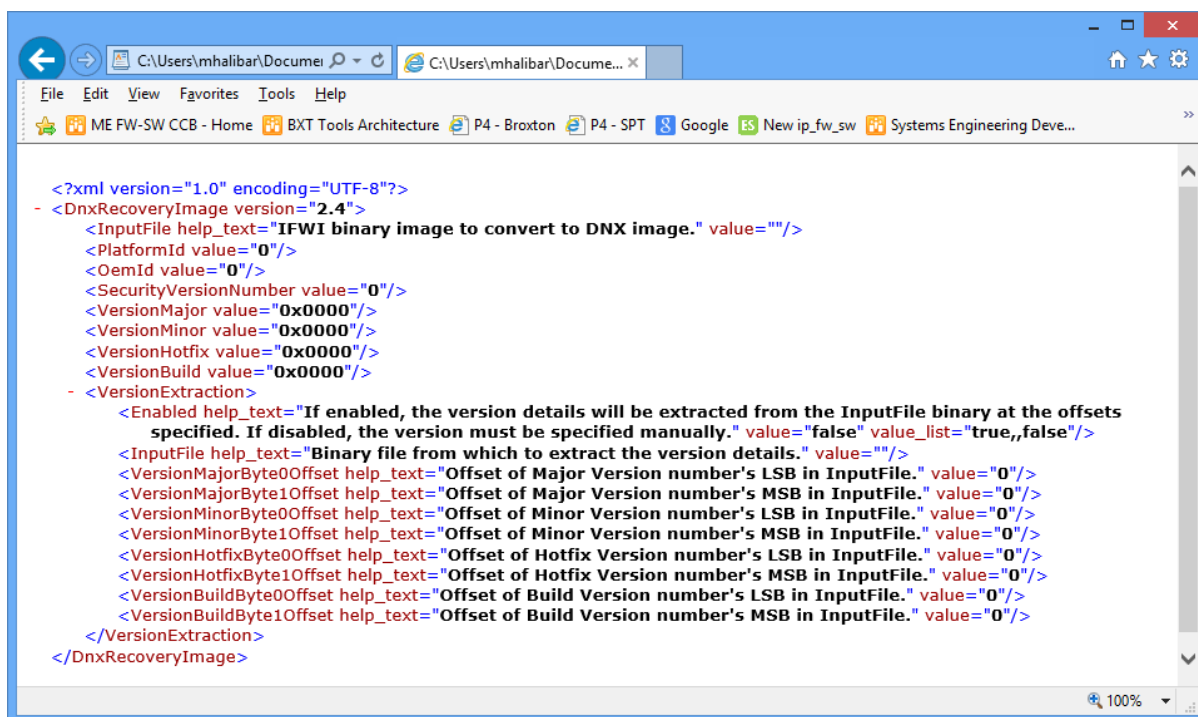
A DnX image can be built from the standard IFWI created by Intel FIT. To do this, you call Intel MEU to generate a template for DnX image creation

```
# meu -gen DnXRecoveryImage
```

This generates an xml template with fields to enter the path to the IFWI binary, and the PlatformID and OemID fields required in a DnX manifest.



Figure 18. DnX xml



Once the xml template has been edited, Intel MEU can be called to create the DnX image

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

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

If the target platform does not have the OEM Key Manifest hash burned in the FPF, the DnX image does not need to be signed (but it does need a manifest).

7.2.2 DnX Image Creation Using Intel FIT

You can use Intel FIT to create a DnX image. Intel FIT interfaces with Intel MEU to do the steps described in the previous section. To use Intel FIT, configure Intel FIT to also build, manifest and sign a DnX image during compilation time. This requires opening Intel FIT Settings dialog, which includes the required fields:

- Path to the private key for signing the DnX image, which is the same key used for signing the OEM Key Manifest. This is only required if the target platform has the OEM Key Manifest hash burned in the FPF
- OemID
- PlatformID



7.3 BIOS Capsule Update

The creation of the capsule is not done via Intel FIT or Intel MEU, but via BIOS tools.

Important

An image used for BIOS capsule update must be signed with the key whose hash appears in the OEM Key Manifest 'IfwiManifest' node. If the platform OEM Key Manifest hash was not burned in the FPF, the update image does not need a signature.





8 Using Intel MEU with Other Signing Tools

8.1 Introduction

Some OEMs will have already existing signing tools and systems, and will want to use Intel® MEU together with them, and not have to integrate with OpenSSL.

This can be supported – however, it is more complex. The steps are as follows

1. Create key pairs and key hashes.
2. Create all manifested binaries.
3. Export manifests
4. Use tool to sign each of the manifests
5. Import the resigned manifests into the binaries.

8.2 Creating Keys and Hashes

Create key pairs and key hashes using the other tools. Creation of the hashes should be done manually, as described in section 3.3.2, but using the alternative tools to do the operations specified for OpenSSL.

8.3 Create Manifested Binaries

Create all manifested binaries, as described in previous chapters. The Intel MEU configuration xml can leave the fields for encryption tool empty, and then the signing steps will be skipped during manifest creation.

8.4 Export Manifests

Use the MEU `-export` function to export the manifest from the 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 need to give 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
4	0x00001A2058	0x0000000378	DNXP.man

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

8.5 Sign Manifests

Use alternative tool to sign the manifest, and enter the crypto information into the manifest.

The manifest header is defined as follows:

Name	Offset	Size (bytes)	Description
Header type	0	4	Must be 0x4
Header Length	4	4	In DWORDs; equals 161 for this version
Header Version	8	4	0x10000 for this version
Flags	12	4	Bit 31: Debug Manifest (manifest is debug signed, not production signed) Bits 0-30: reserved, must be 0
Vendor	16	4	0x8086 for Intel
Date	20	4	yyyymmdd in BCD format
Size	24	4	In DWORDs, size of entire manifest (header + extensions). Maximum size is 2K DWORDs (8KB).
Header ID	28	4	Magic number. Equals "\$MN2" for this version
Reserved	32	4	Must be 0
Version	36	8	Major, minor, hotfix, build
Security Version Number	44	4	SVN, least significant byte used to derive keys
Reserved	48	8	Must be 0
Reserved	56	64	Must be 0
Modulus Size	120	4	In DWORDs; 64 for pkcs 1.5-2048
Exponent Size	124	4	In DWORDs; 1 for pkcs 1.5-2048



Name	Offset	Size (bytes)	Description
Public Key	128	256	
Exponent	384	4	
Signature	388	256	RSA signature of manifest. 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, and then the 3 'crypto' fields in the manifest header populated with the key, exponent and signature.

No other fields in the manifest should be changed.

8.6 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, which is passed as an input parameter. If the binary supports multiple manifests (e.g. a full IFWI 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>
```

§



9 Common Bring Up Issues and Troubleshooting Table

9.1 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 of the Intel MEU tool. Confirm that both files have been modified properly.
Rebuild of image requires SMIP private key	Any change to an image will require a full rebuild, including the SMIP, which requires the SMIP private signing key. If the image is a customer image, Intel debug teams will need the signing key to rebuild the image, or request the customer to rebuild.

§