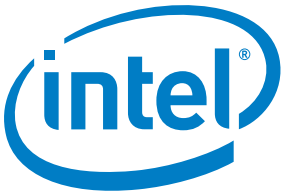


Intel® Gateway Solutions for the Internet of Things – Development Kit – DK300 – Secure Boot

ODM and User's Implementation Guide

July 2014

Document Number: 547808

By using this document, in addition to any agreements you have with Intel, you accept the terms set forth below.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

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.

Code Names are only for use by Intel to identify products, platforms, programs, services, etc. (“products”) in development by Intel that have not been made commercially available to the public, i.e., announced, launched or shipped. They are never to be used as “commercial” names for products. Also, they are not intended to function as trademarks.

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

Intel, Intel Atom, Intel Core, Intel® Hyper-Threading Technology, Intel® Trusted Execution Technology, Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Wind River is a trademark of Wind River Systems, Inc.

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

Contents

1 About This Document 6
- 1.1 Objective 6
- 1.2 Target Audience 6
- 1.3 Terminology 6

2 Secure Boot Enablement 7
- 2.1 High-Level Flow 7
- 2.2 Detailed Secure Boot Flow 8

3 Creating the Secure Boot Image and Flash Target Device 11
- 3.1 Splitting UEFI Stages, Producing Stage Hashes, Creating Data Region 11
- 3.2 Creating the Secure Boot Firmware Image and Flashing the Target Device 13
  - 3.2.1 Creating the 5 MB BIOS with Secure Boot Manifest 14
  - 3.2.2 Creating the SPI Flash Binary 16
  - 3.2.3 Flashing the SPI Image on the Target Device Using UEFI shell 22
  - 3.2.4 Flashing the SPI Image onto the Target Device Using Dediprog 25

4 Programming the Field Programmable Fuses (FPF) 27

5 Implementing UEFI Secure Boot (Stage 3) 29
- 5.1 Creating Owner Public and Private Keys 30
- 5.2 Creating Vendor Public and Private Keys 31
- 5.3 Signing Bootloader, Kernel, RootFS, RPM Packages 32
  - 5.3.1 Signing the Bootloader 32
  - 5.3.2 Signing the Kernel 33
  - 5.3.3 Signing RootFS 34
  - 5.3.4 Signing RPM Packages 35
  - 5.3.5 Deploying RootFS Image 36

6 Configuring UEFI for Secure Boot 37
- 6.1 Change the BIOS Settings 37
- 6.2 Enable Secure Boot 38
- 6.3 Test Secure Boot 38

Figures

Figure 1. SPI Flash Region Definition 8
Figure 2. Secure Boot Flow 9

Tables

Table 1. Terminology 6
Table 2. High-Level Boot Flow 7
Table 3. Configuration Order 8
Table 4. Required Software 11
Table 5. Files to Provide to End-User 12
Table 6. Required Software Tools 13
Table 7. Required Files 14
Table 8. Intelligent Device Platform Runtime Image Files 29
Table 9. Software Tools Required for UEFI Secure Boot 29
Table 10. Intelligent Device Platform Runtime Image Files 30
Table 11. Vendor Key SST Commands 31
Table 12. Bootloader Signing SST Commands 32
Table 13. Kernel Signing SST Commands 33
Table 14. RootFS Signing SST Commands 34
Table 15. RPM Signing SST Commands 35

Revision History

Date	Revision	Description
July 2014	1.0	First release

§

1 About This Document

1.1 Objective

This document provides instructions to implement Secure Boot for the Intel® Gateway Solutions for the Internet of Things – Development Kit – DK300 platform.

1.2 Target Audience

This publication is targeted to ODMs and end users for the Intel® Gateway Solutions for the Internet of Things – Development Kit – DK300.

1.3 Terminology

In this publication:

- "Target Device" refers to the Intel® Gateway Solutions for the Internet of Things – Development Kit – DK300
- "Host System" refers to computer that is used to contact and work with the Target Device.

Table 1. Terminology

Term	Description
IBB	Initial Boot Block
ODM	Original Design Manufacturer
OEM	Original Equipment Manufacturer
Intel® TXE	Intel® Trusted Execution Engine
UEFI	Unified Extensible Firmware Interface
WiFi*	Wireless-Fidelity

§

2 Secure Boot Enablement

2.1 High-Level Flow

The figure below shows the high level secure boot flow. After power on:

- The Intel® Trusted Execution Engine verifies the authenticity of the Secure Boot Manifest.
- The Secure Boot Manifest verifies the authenticity of Stage 1.
- Stage 1 verifies the authenticity of Stage 2.
- During execution of Stage 2, the UEFI key database verifies the bootloader
- The bootloader verifies the authenticity of the kernel.

Table 2. High-Level Boot Flow

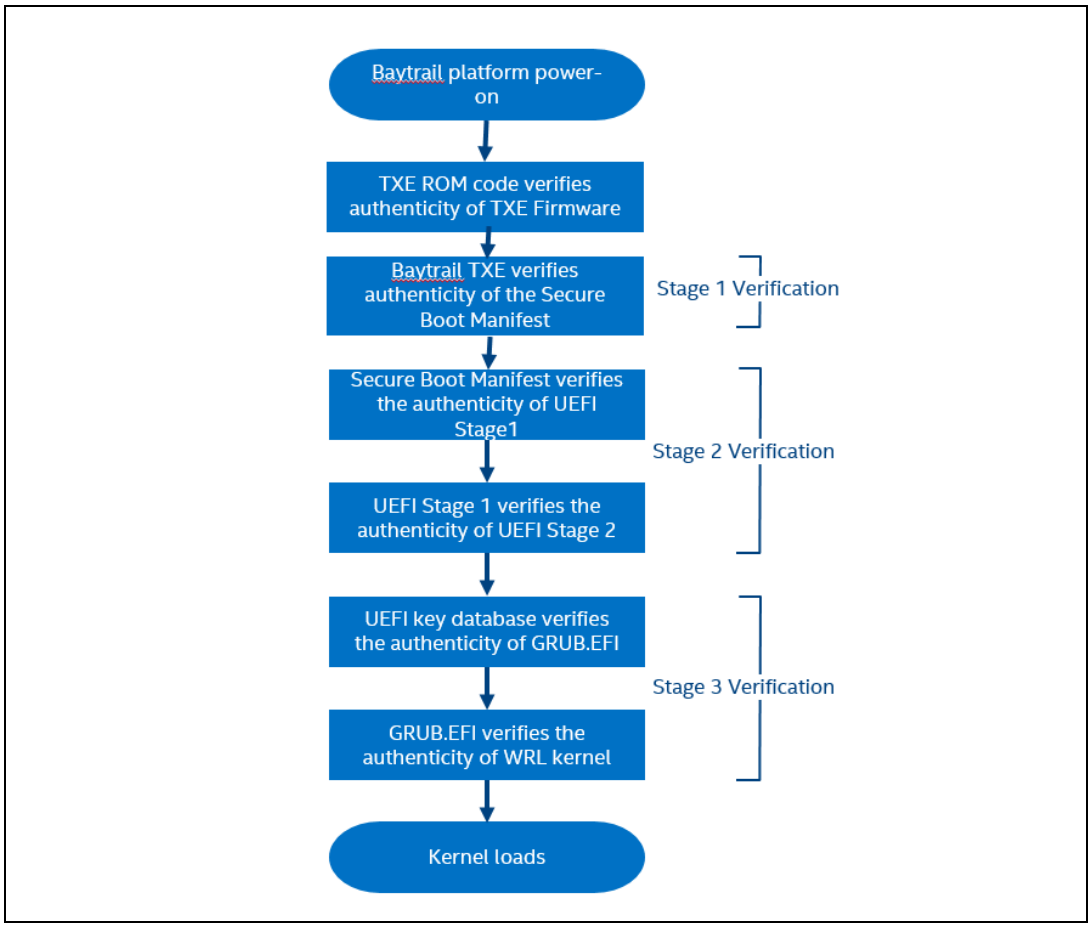


Table 3. Configuration Order

Order of Configuration	Stage Name	Configured by
First	Stage 2 Verification	ODM or end user
Second	Stage 1 Verification	ODM or end user
Third	Stage 3 Verification	End user

2.2 Detailed Secure Boot Flow

To understand the secure boot flow, it is important to understand the SPI flash layout. As shown in the figure below, the BIOS binary in SPI flash contains the following components:

- BIOS Stage 1 (Initial Boot Block)
- BIOS Stage 2
- Secure Boot Manifest
- Intel® Trusted Execution Engine (Intel® TXE) firmware

Figure 1. SPI Flash Region Definition

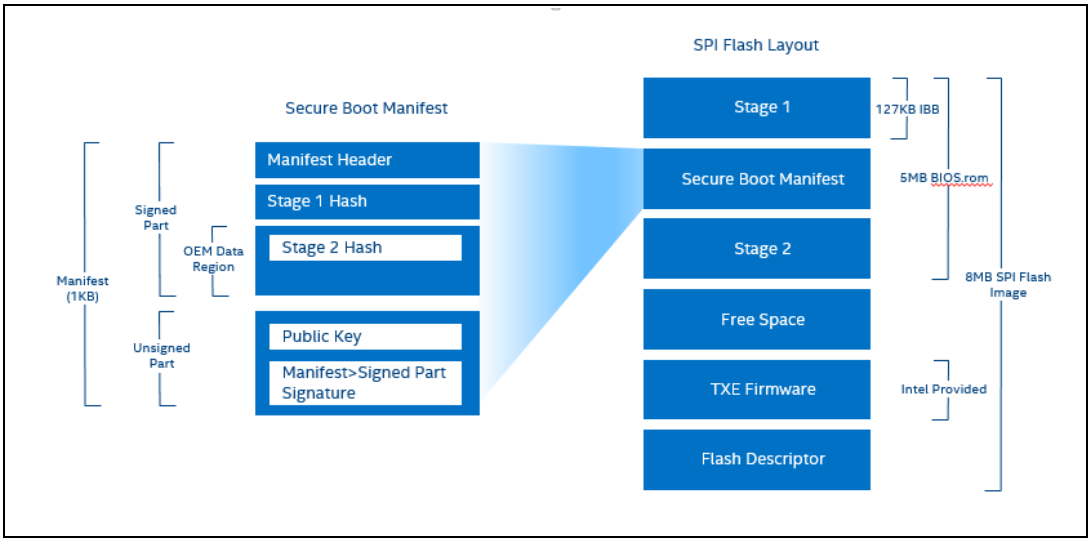
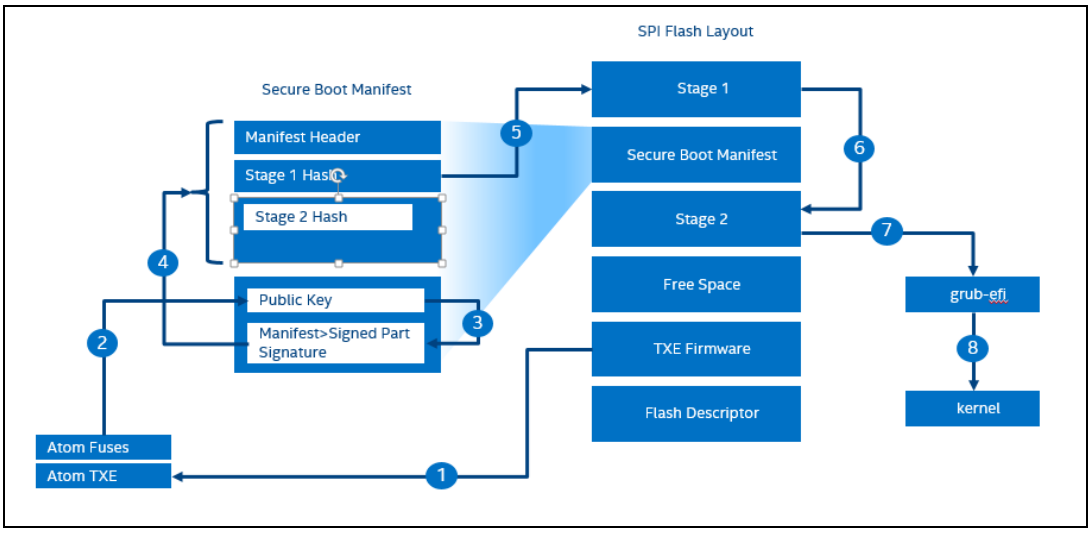


Figure 2. Secure Boot Flow



Using the figures above as a reference, the secure boot flow is as follows:

1. When the Intel® Atom™ processor-based gateway is powered-on, the CPU is held in reset and does not execute code. The Intel® Trusted Execution Engine ROM code verifies the authenticity of Intel® Trusted Execution Engine firmware. The Trusted Execution Engine in the Intel® Atom™ processor fetches and executes the Intel® Trusted Execution Engine firmware resident on the SPI flash.
2. The Intel® Trusted Execution Engine firmware reads the internal Intel® Atom™ processor fuses. This is the SHA-256 hash of the RSA public key stored in the unsigned region of the secure boot manifest. The Intel® Trusted Execution Engine firmware performs a SHA-256 hash of the RSA public key stored in the unsigned region of the secure boot manifest. The Intel® Trusted Execution Engine firmware compares the calculated hash value against the value stored in the processor fuses. If no match is found, execution is halted.
3. The signature of the signed part of the secure boot manifest is stored in the Manifest's Unsigned Part. See Figure 1. This signature is decrypted with the RSA public key that was verified in Step 2. The decryption result is the SHA-256 hash of the Manifest's Signed Part.
4. The Intel® Trusted Execution Engine firmware calculates the SHA-256 hash of the Manifest's Signed Part and compares the result with the decrypted hash from Step 3. If no match, execution will halt. Since the IBB and UEFI Stage 2 hashes are included in the Manifest's Signed Part, successful verification of this step guarantees the IBB and BIOS Stage 2 hashes are authentic and un-altered.
5. The Intel® Trusted Execution Engine firmware calculates the SHA-256 hash of the Initial Boot Block. The calculated hash is compared to the authenticated IBB hash in the Manifest. If the hashes are identical, then the IBB is authenticated and the power management controller releases the CPU from reset. The CPU fetches and executes the IBB, which performs basic initialization of the platform. This includes memory initialization.

6. The IBB code calculates the SHA-256 hash of the UEFI Stage 2 and compares the result against the UEFI Stage 2 hash that was authenticated in Step 4. If the hashes are not identical, execution is halted.
7. The UEFI Stage 2 calculates the hash of `grub.efi` and compares it to the keys that are enrolled in the UEFI key database. If the hashes are not identical, execution is halted.
8. `grub.efi` decrypts the signature appended to the Linux kernel and verifies the integrity. If verification passes, the system continues to load and execute the kernel. Otherwise, execution is halted.

3 Creating the Secure Boot Image and Flashing Target Device

The creation of the secure boot image is divided into two steps. The first must be completed by an ODM. The second can be completed by either an ODM or an end user:

- ODM only: Split UEFI stages, produce stage hashes, and create the data region.
- End user or ODM: Create the secure boot firmware image and flash it onto the Target Device.
  - ODM: If you generate the key pair and you are keeping the private key secure, then you can optionally complete this step for your end user(s).
  - End user: If you want to generate your own key pair, then your ODM will complete the first step and provide you with the files you need to create the secure boot firmware image.

3.1 Splitting UEFI Stages, Producing Stage Hashes, Creating Data Region

Note: Only an ODM should complete the steps in this section.

The following software tools are required to complete these steps. These software tools listed below are provided by AMI and use Microsoft* Windows 7 or 8. The files do not need to be saved into any specific directory.

Table 4. Required Software

Software Tool	Description
split.exe	Splits binary files. Used to separate UEFI Stages
CryptoCon.exe	Hashes UEFI Stages
insert.exe	
ml.exe	Microsoft* Linker
link.exe	
GenFW.exe	

1. The ORG.ROM file is provided by the BIOS vendor. It is 5 MB in size. Execute the following commands to split it into IBB and UEFI Stage 2.
   - `split -f ORG.ROM -s 5115904 -o FD_NON_PRE_BB.fd -t FD_PRE_BB.fd`
   - `split -f FD_PRE_BB.fd -s 1024 -o FD_PRE_BB_1K.fd -t FD_PRE_BB_127K.fd`
2. Use the following commands to generate the IBB and UEFI Stage 2 hashes:
   - `split -f ORG.ROM -s 331776 -o ORG.331776 -t ORG.4911104`
   - `split -f ORG.4911104 -s 4251648 -o ORG.4251648`
   - `split -f ORG.ROM -s 4190208 -o ORG.4190208 -t ORG.left`
   - `split -f ORG.left -s 393216 -o FV_BB.Fv -t ORG.left2`
   - `CryptoCon.exe -h2 -f ORG.4251648 -o HashSecondStageKey.bin`
   - `CryptoCon.exe -h2 -f FV_BB.Fv -o HashFvBbKey.bin`
3. Use the following commands to generate the OEM data region. See Figure 2:
   - `insert.exe CreateIncFromBin HashSecondStageKey.inc HashSecondStageKey.bin`
   - `insert.exe CreateIncFromBin HashFvBbKey.inc HashFvBbKey.bin`
   - `ml /c /nologo /Fo GenBiosImageInfo.obj GenBiosImageInfo.asm`
   - `link /NOENTRY /FIXED /DLL GenBiosImageInfo.obj /OUT:GenBiosImageInfo.dll`
   - `genfw --exe2bin GenBiosImageInfo.dll -o GenBiosImageInfo.bin`
   - `split -f GenBiosImageInfo.bin -s 0x64 -o BiosImageInfo.bin`
4. If your end user will be creating the secure boot firmware image, provide the following files:

Table 5. Files to Provide to End-User

Filename	Description
FD_PRE_BB_127K.fd	IBB (127KB IBB)
FD_NON_PRE_BB.fd	BIOS Stage 2
BiosImageInfo.bin	OEM Data Region
ORG.ROM	The 5 MB BIOS ROM from the BIOS vendor

3.2 Creating the Secure Boot Firmware Image and Flashing the Target Device

Either the end-user or the ODM can complete the steps in this section.

The steps below guide you through creating a secure boot firmware image with the hardware root that will be flashed into the SPI flash device of the Intel® Gateway Solutions for the Internet of Things – Development Kit – DK300.

Note: An Intel CNDA is required to access these tools. Contact your ODM for information.

Table 6. Required Software Tools

Software Tool	Operating System	Provided by	Description
FLAMInGo.exe	Windows 7 or higher	Intel	Hashing and secure boot manifest creation
FusesConfiguration.bat	Windows 7 or higher	Intel	Sets up the manifest generation tool environment
Simplesigner.exe	Windows 7 or higher	Intel	Signs an input file with a private key and provides an output signature. An ODM will need to develop their own tool.
OpenSSL	Windows 7 or higher, Linux	Open source	Signs binaries, creates key pairs, wraps private keys.
FITC.exe	Windows 7 or higher	Intel	Combines descriptor, BIOS, PDR, and Intel Intel® Trusted Execution Engine FW binaries into one image.
FPT/FPT64	Linux, EFI, Windows 7 or higher	Intel	Flash Programming Tool. Programs the flash memory of individual regions or the entire flash device and one-time programming Intel® Trusted Execution Engine fuses of Intel® Atom™ processor.

In addition to the software tools, the following files are needed. The files do not need to be saved into any specific directory.

Table 7. Required Files

Filename	Description	Provided By
FD_PRE_BB_127K.fd	IBB (127KB IBB)	ODM
FD_NON_PRE_BB.fd	UEFI Stage 2	ODM
BiosImageInfo.bin	OEM Data Region	ODM
ORG.ROM	The 5 MB BIOS ROM that originates from the BIOS vendor	ODM
TXE_Region_3MB.7z	The 3 MB Intel® Trusted Execution Engine firmware	Intel
oFpfMirrorNvarValues.txt	Used to generate optional FPT mirror values	Intel

3.2.1 Creating the 5 MB BIOS with Secure Boot Manifest

1. Generate a public and private key pair to sign and verify the Secure Boot Manifest. You can modify the `openssl` parameters based on your security policy. Use the command:

Note: Change the command to include the full path to the `openssl.cnf` file. For example, `-config "c:\OpenSSL-Win64\bin\openssl.cnf"`

```
openssl req -batch -x509 -nodes -days 9000 -newkey rsa:2048 -keyout
"privatekey.pem" -out "publickey.pem" -config "openssl.cnf"
```

The output of this command is two files placed into your working directory: `privatekey.pem` and `publickey.pem`.

2. Change the encoding of the private key to PKCS#8. Use the following commands:

```
openssl pkcs8 -topk8 -in privatekey.pem -out privatekey-pk8.pem -nocrypt
```

Rename `privatekey-pk8.pem` to `privatekey.pem` The output of this command is `privatekey.pem` in PKCS#8 encoding.

3. Hash the public key using the FLAMInGo tool from Intel. Use the following command:

```
FLAMInGo.exe HashKey publickey.pem publickeyhash.txt
```

The output of this command is `publickeyhash.txt` placed into your working directory

4. Create the Intel® Trusted Execution Engine fuse mirror values that will be programmed into the Field Programmable Fuses of the Intel® Atom™ processor. Use the following command and batch file:

```
FusesConfiguration.bat publickeyhash.txt oFpfMirrorNvarValues.txt
FpfMirrorNvarValues.txt
```

The output of this command is `FpfMirrorNvarValues.txt` placed into your working directory

5. Create the Manifest structure and the hash of the Manifest Signed Part. Use the following command:

```
FLAMInGo.exe SBManCreate FpfMirrorNvarValues.txt AmiManifest
FD_PRE_BB_127K.fd 2 publickey.pem -OEMDataFile BiosImageInfo.bin
```

The output of this command is two files placed into your working directory: `AmiManifest_SB_config.xml` and `AmiManifest_SB_hash.bin`

6. Sign the Manifest's hash with your private key. The following command is provided as a sample. Signer of the manifest must use their own tool.

```
Simplesigner.exe privatekey.pem AmiManifest_SB_hash.bin
AmiManifest_SB_signature.bin
```

The output of this command is `AmiManifest_SB_signature.bin` placed into your working directory.

7. Create the Secure Boot Manifest. Inputs to this process are the fuse mirror values, manifest xml structure and the manifest signature. Use the following command:

```
FLAMInGo.exe SBManComplete FpfMirrorNvarValues.txt AmiManifest
AmiManifest_SB_signature.bin
```

The output of this command is `AmiManifest_SB_Manifest.bin` placed into your working directory

8. From the Windows CMD window merge the BIOS Stage 2 with the Manifest and IBB. Use the following command:

```
copy /Y /B FD_NON_PRE_BB.fd+AmiManifest_SB_manifest.bin BIOS.ROM
```

The output of this command is `BIOS.ROM` placed into your working directory. The file is 5 MB in size.

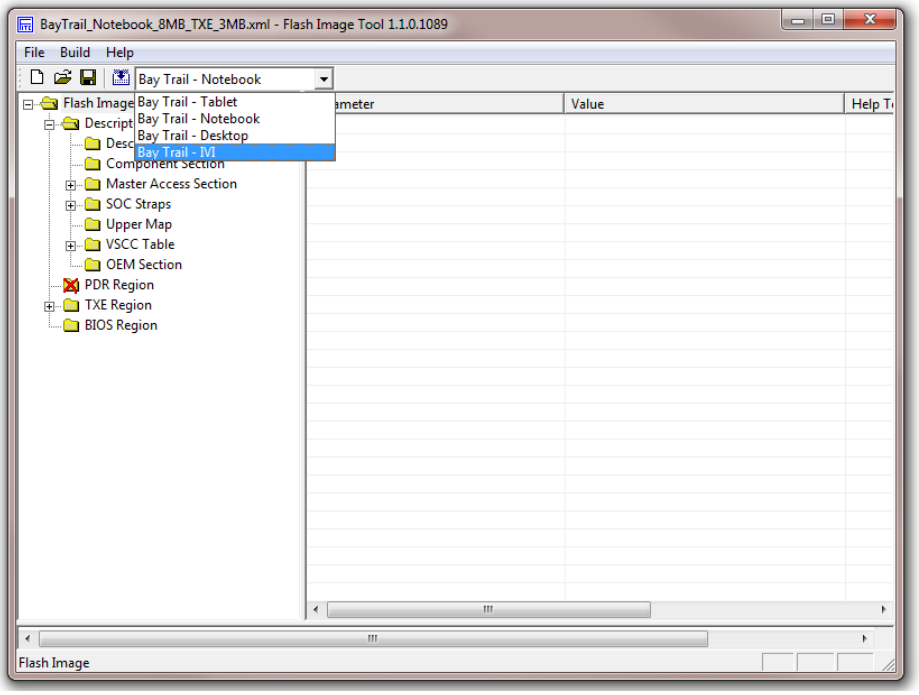
3.2.2 Creating the SPI Flash Binary

The following steps guide you through merging `BIOS.ROM` with the Intel® Trusted Execution Engine firmware. This is the binary that will be programmed into the Target Device's SPI flash device.

1. Unzip Intel_Tools.zip
2. The BIOS.ROM file was generated in the previous set of steps. It is in your working directory. Copy it to `\Intel_Tools\Flash_Image_Tool`
3. Unzip `TXE_Region_3MB.7z` into `\Intel_Tools\Flash_Image_Tool`
4. Copy `FpfMirrorNvarValues.txt` into `\Intel_Tools\Flash_Image_Tool`
5. Use the following command to run the Flash Image Tool:

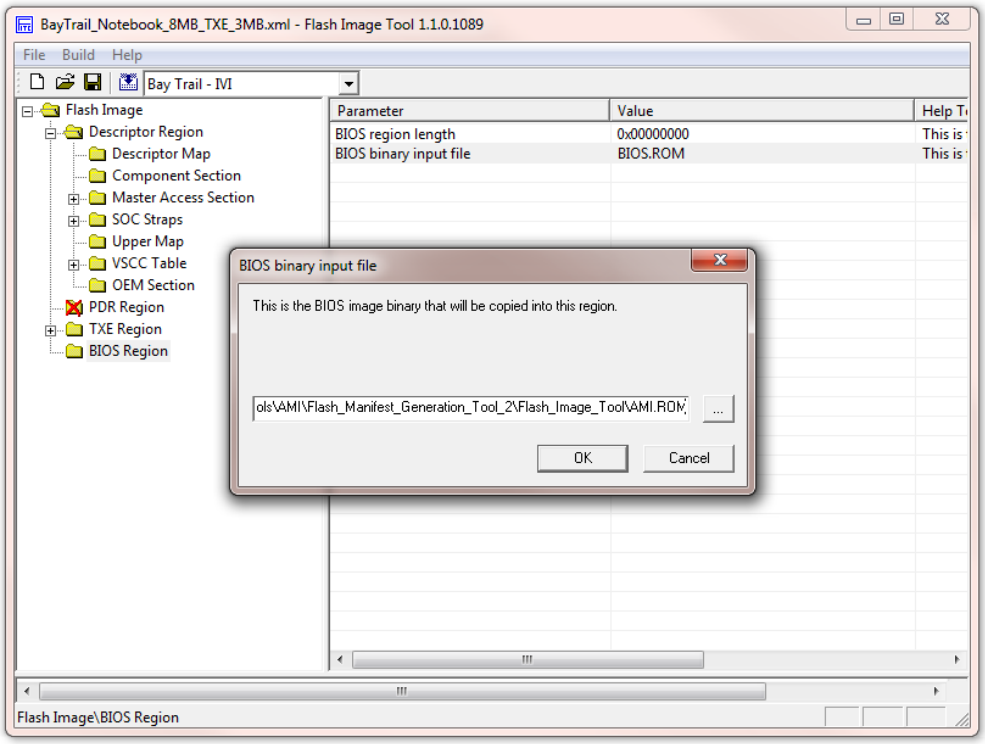
```
\Intel_Tools\Flash_Image_Tool\fitc.exe
```

6. When the Flash Image Tool opens, select **Baytrail-IVI** from the drop-down. See the following figure:



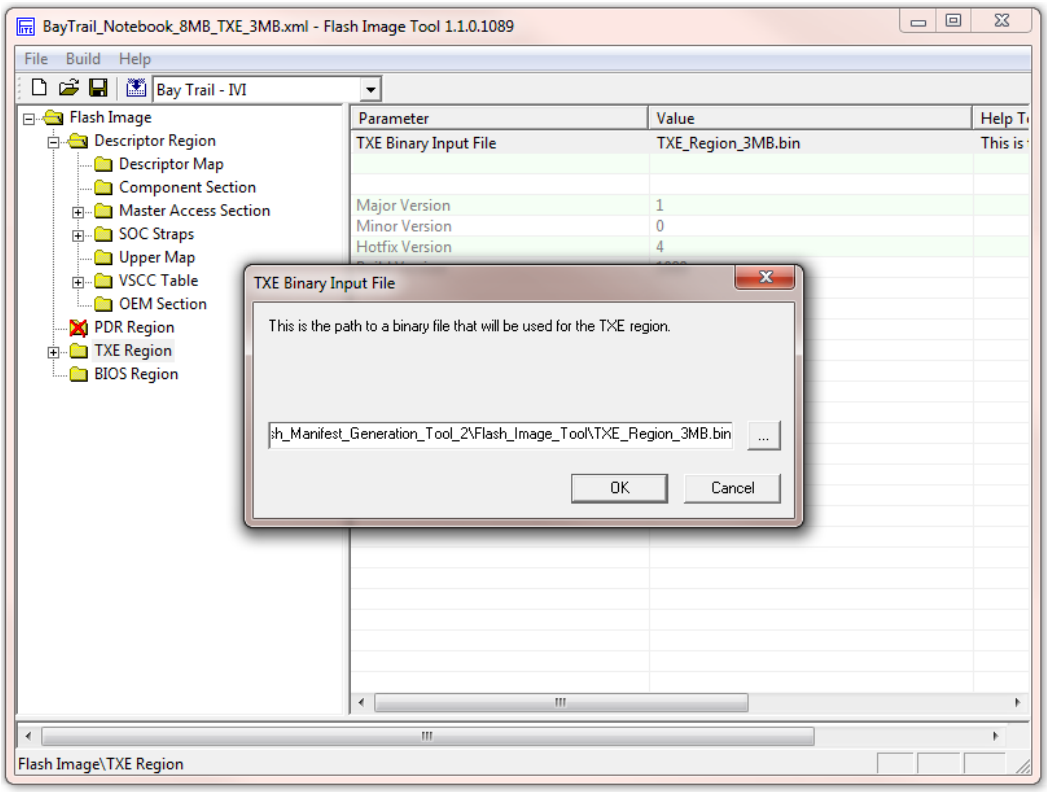
7. Select **BIOS Region** from the left pane.
8. Select **BIOS binary Input File** from the right pane.

9. Double click the **Value** column and browse to select `BIOS.ROM` to load the BIOS region. See the following figure.



10. Select **TXE Region** from the left pane.
11. Select **TXE Binary Input File** from the right pane.

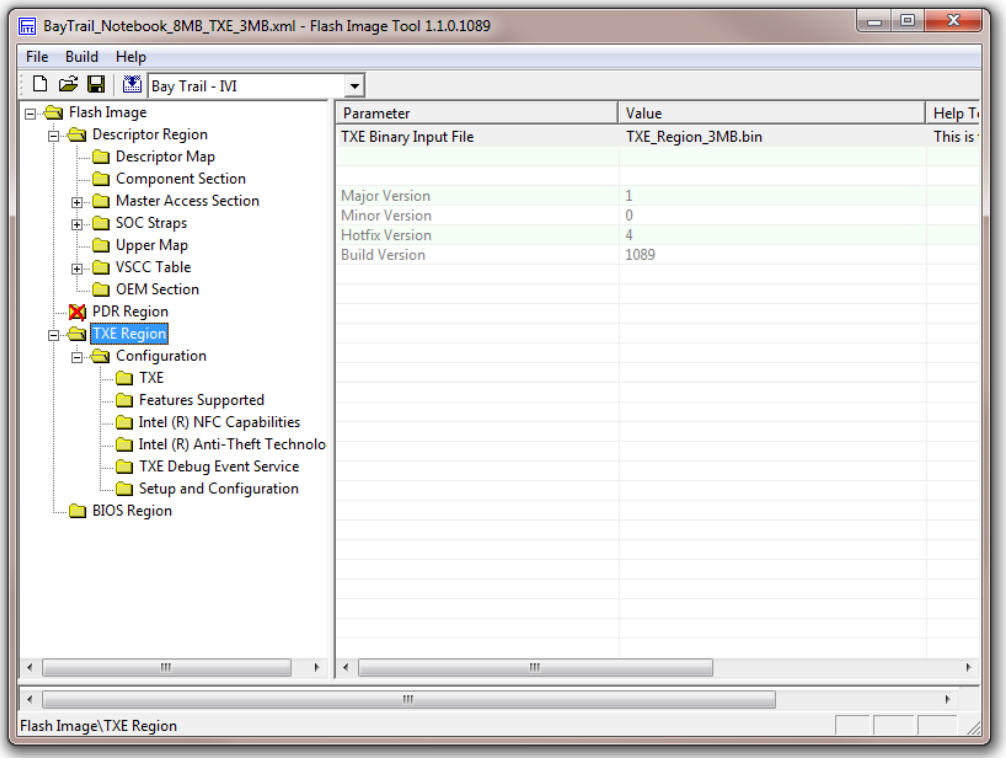
12. Double click the **Value** column and browse to select `TXE_Region_3MB.bin`. The Intel® Trusted Execution Engine region loads. See the following figure.



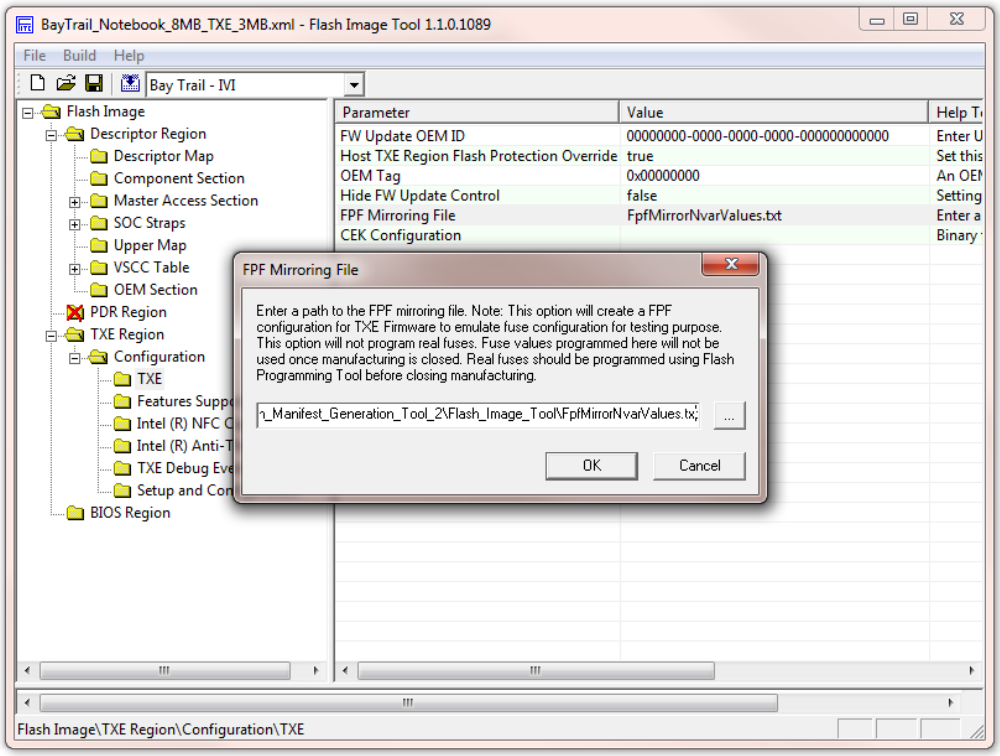
13. Optional: The following steps load the FPF configuration file for FPF mirroring. Use these steps only if FPF Mirroring will be used instead of HW FPF programming, such as in a non-production environment. If you are not using FPF Mirroring, skip to step 17.

Note: In a production environment the FPF Mirroring Configuration File should not be included in the Stitching process.

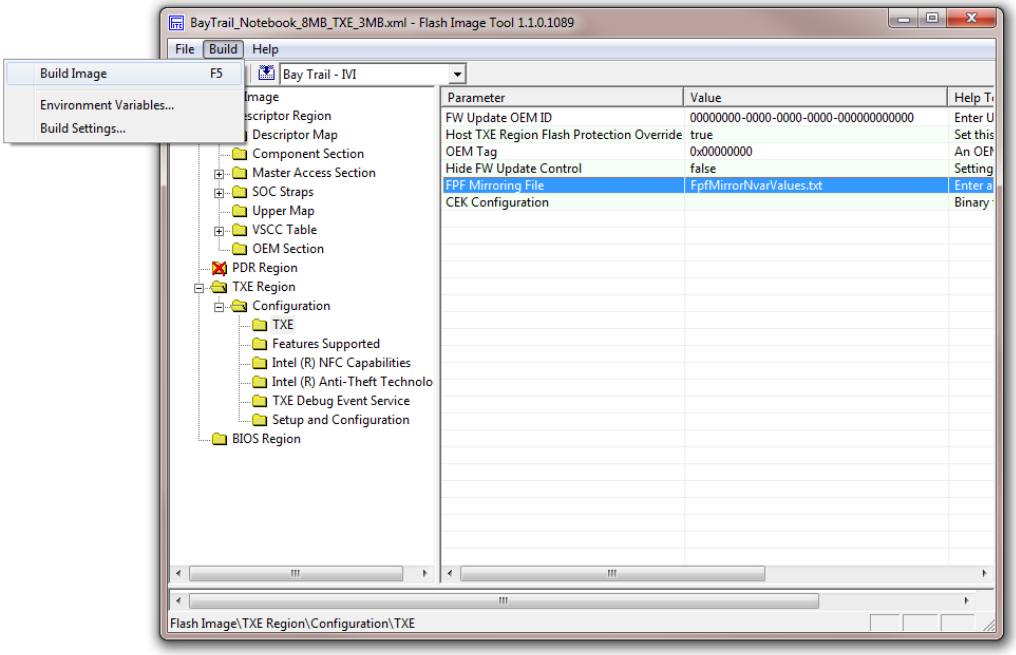
Click the **+** next to **TXE Region** in the left pane and click the **+** next to **Configurations**. See the following figure.



14. Select **TXE** from the left pane.
15. Select **FPF Mirroring File** from the right pane.
16. Double click the **Value** column and browse to select `FpfMirrorNvarValues.txt`. See the following figure.



17. To complete the stitching process, build the binary image. Click **Build** -> **Build Image**. See the following figure.



This produces the 8M SPI binary image (`outimage.bin`) to flash onto the Target Device.

3.2.3 Flashing the SPI Image on the Target Device Using UEFI shell

The UEFI shell can be programmed using the UEFI shell or dediprog. This section uses the UEFI shell. This process requires a working BIOS on the Target Device. If the Target Device does not have a working BIOS, see Section 3.2.4, Flashing the SPI Image onto the Target Device using dediprog.

Note: Upon completing the following step, all contents of the USB flash drive will be deleted. Back up any necessary content before proceeding.

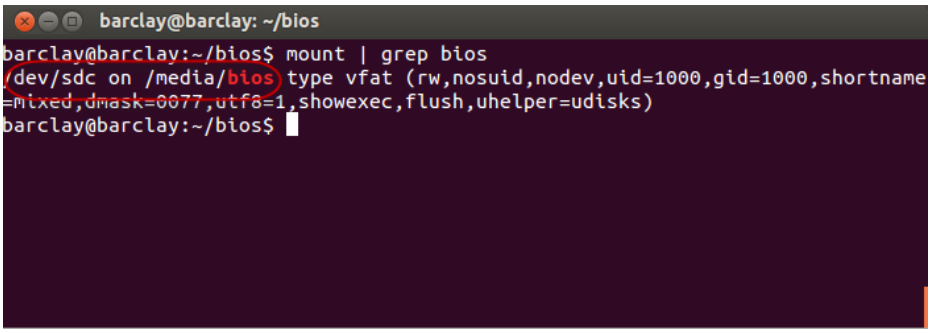
1. Use the following command with a 1 GB or larger USB flash drive:

```
sudo mkdosfs -n 'bios' -I /dev/sd?
```

2. Remove and reinsert the USB flash drive. It will mount automatically into `/media/bios`. Use the following command to confirm the mount point:

```
mount
```

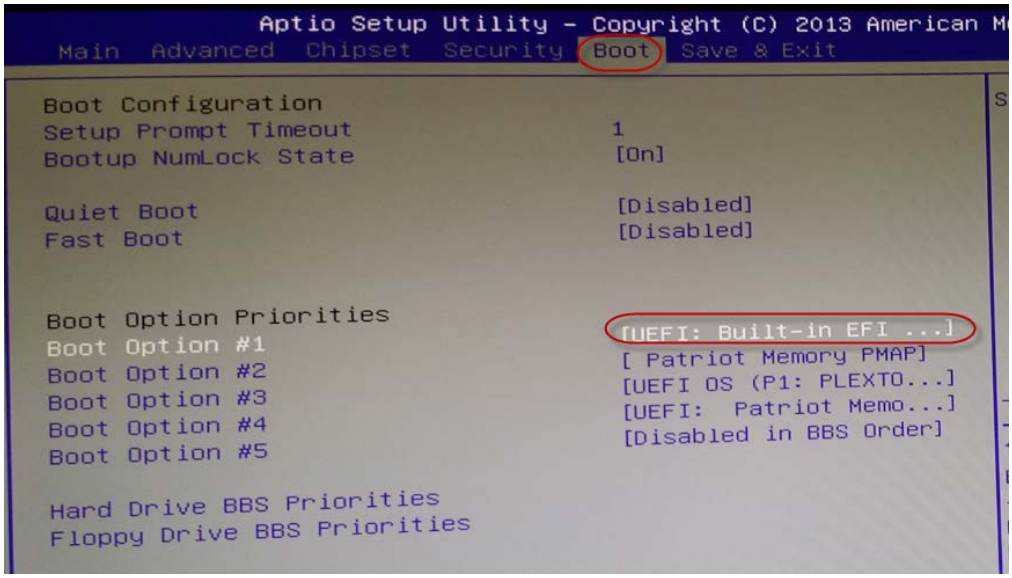
In the example shown below, `/dev/sdc` is mounted on `/media/bios`. See the text circled in red.



3. Use the command below to copy the following files from the Host System to the USB flash drive:
   - `fparts.txt`
   - `fpt64.efi`
   - `outimage.bin`

```
cp fp* outimage.bin /media/bios
```

4. Unmount the USB flash drive.
5. Plug the USB flash drive into the Target Device.
6. Power on the Target Device and immediately press the DEL key to load the BIOS Setup screen.
7. Under the **Boot** tab, select **UEFI** for **Boot Option #1** as shown in the following figure.



8. Press the right arrow key to select **Save and Exit**. The Target Device will boot into EFI.
9. At the EFI prompt, enter the fp0 partition on the USB flash drive and then confirm the BIOS files are present (`fparts.txt`, `fpt64.efi`, `outimage.bin`). Use the following commands:

```
fp0:
ls
```

Note: Do not continue before confirming you are in the location with the necessary files. If necessary, change to the fp1 partition and look again for the files. To change to the fp1 partition to look again, use the commands:

```
fp1:
ls
```

10. Complete the BIOS upgrade with the following command. Watch for errors.

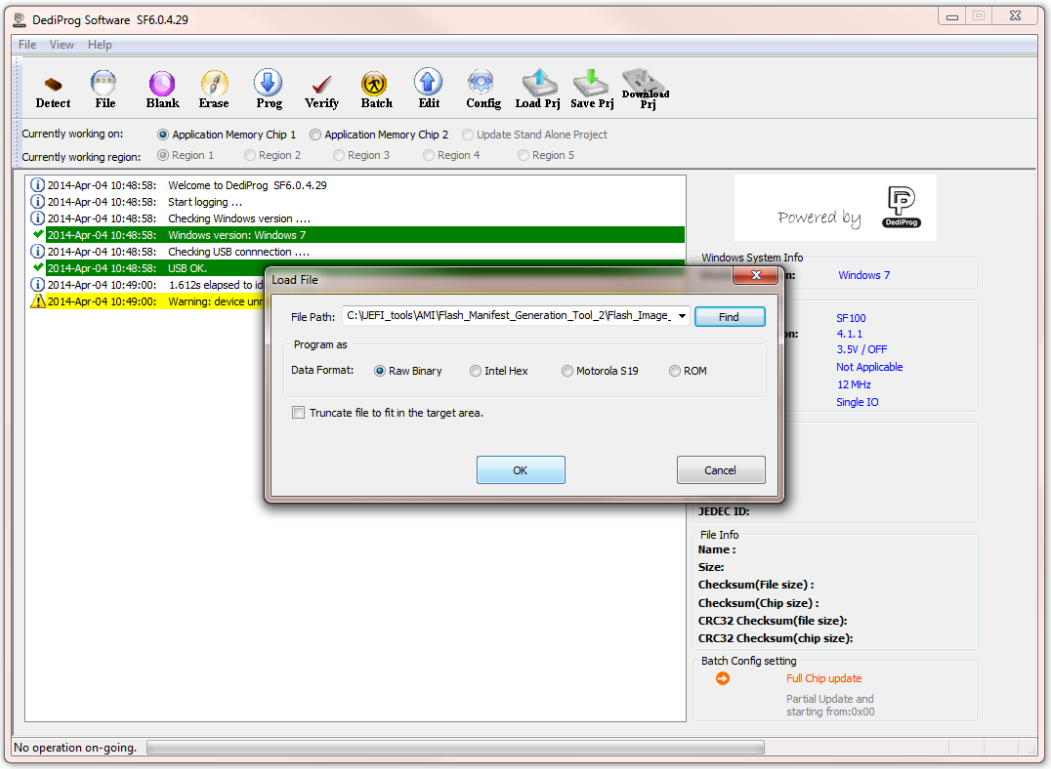
```
fpt64.efi -F outimage.bin
```

3.2.4 Flashing the SPI Image onto the Target Device Using Dediprog

Note: If you successfully flashed the SPI Image onto the Target Device using the UEFI shell, disregard this section.

The next steps guide you through flashing the SPI image using dediprog if your original BIOS is not functioning and you cannot access the UEFI shell.

1. Connect the Dediprog SF100 programmer to the USB port of your Host System.
2. Connect the 8-pin header to the Target Device.
3. Execute DediProg Engineering software.
4. Click **File** and browse to select `outimage.bin`. See the following figure.



5. Click **Batch** to begin programming to the SPI.
6. Once programming is done, click **Verify** to verify the checksum of the written image in SPI and in the file buffer.
7. If checksum verification is ok then disconnect the Dediprog programmer header from the Target Device.
8. Power up the Target Device.

4 Programming the Field Programmable Fuses (FPF)

Warning: The process in this section is irreversible and can be completed only one time. Errors will render the Target Device permanently inoperable. Proceed with caution.

1. Boot the Target Device to the EFI shell.

Warning: Upon completing the following step, all contents of the USB flash drive will be deleted. Back up any necessary content before proceeding.

2. Use FAT32 format to prepare a USB flash drive.
3. Copy `\Intel_Tools\Flash_Programming_Tool` to the USB flash drive.
4. Insert the USB flash drive into the Target Device.
5. Map the USB flash drive to the EFI Shell. Use the following command:

```
map -r
```

6. Change the path to the USB flash drive (for example: `Shell>fs0:`)
7. Change to the EFI directory. Use the following command:

Note: In the following command, replace EFI with EFI32 if this is a 32-bit shell.

```
cd \Intel_Tools\Flash_Programming_Tool\EFI
```

8. Read the default values in FPF. Use the following commands:

```
fpt64.efi -READFPF OEM_KEY_HASH_1

fpt64.efi -READFPF SECURE_BOOT_EN

fpt64.efi -READFPF Global_Valid
```

9. Program and Read back the OEM KEY HASH values. Use the following command.

Note: In the command below, replace <OEM hash value> with the actual hash value. For example: `fpt64.efi -WRITEFPF OEM_KEY_HASH_1 -v 0x2D8CE4A658C6A45D2B64C7FB3D73A31893EB58274BB383274BBAFDD355E55016`

Warning: Do not use the sample value in the following programming command, You must use your actual hash value.

```
fpt64.efi -WRITEFPF OEM_KEY_HASH_1 -v <OEM hash value>

fpt64.efi -READFPF OEM_KEY_HASH_1
```

5 Implementing UEFI Secure Boot (Stage 3)

UEFI Secure Boot is the technology name used by the UEFI Forum to describe UEFI verification of UEFI applications. Any reference here to UEFI Secure Boot is referring to the process of the UEFI Stage 2 verification of the grub.efi, a UEFI application. UEFI Secure Boot is used to verify the authenticity of the grub.efi boot loader. Once grub.efi has been authenticated, grub.efi verifies the authenticity of the kernel.

Note: This section assumes you have completed the build of your Wind River® Intelligent Device Platform runtime image. The build process creates the files listed below. For more information on the build process, see the *Wind River Intelligent Device Platform Programmer's Guide XT 2.0.*

Table 8. Intelligent Device Platform Runtime Image Files

Name	Description	Location
bzImage-intel-atom-baytrail.bin	Kernel image file	$Projdir/export/image/
bzImage-initramfs-intel-atom-baytrail.bin	Kernel image file with initramfs	$Projdir/export/image/
grub.efi	Bootloader image file	$Projdir/build/grub-efi-0.97-r4/image/boot/grub/
grub.conf	Bootloader configure file	$Projdir/build/grub-efi-0.97-r4/image/boot/grub/
wrlinux-image-glibc-idp-intel-atom-baytrail.tar.bz2	IDP Root File System tar ball	$Projdir/export/image/
wrlinux-image-glibc-idp-intel-atom-baytrail-dist-srm.tar.bz2	IDP Root File System signed by SST tar ball	$Projdir/export/image/
wrlinux-image-glibc-idp-intel-atom-baytrail.ext3	IDP Root File System block file	$Projdir/export/image/
wrlinux-image-glibc-idp-intel-atom-baytrail-srm.ext3	IDP Root File System signed by SST block file	$Projdir/export/image/

The tools listed below are required for the UEFI Secure Boot implementation process.

Table 9. Software Tools Required for UEFI Secure Boot

Tool	Operation System	Provided by	Description
SST	Wind River Linux	Wind River IDP	Signing tool for grub.efi, kernel, RootFS and RPM.

5.1 Creating Owner Public and Private Keys

This section guides you through creating the necessary chain of certificates and keys. The root of this chain is an RSA pair called Owner-Cert and Owner-Private. The Owner can bring a private key and use SST to generate a new public cert, based on the existing private key.

The following table provides a description of the SST options.

Table 10. Intelligent Device Platform Runtime Image Files

Option	Description	Current Value	Default Value
verbose	Open or close the signing trace. Value can be yes or no.	no	no
name	User defined name for the role.	ownerE	Name of the role
output-dir	The output directory where you can find your private key and CA certificate	./outputE	SST current directory (“.”)
machine	Target architecture.	intel_atom_baytrail	intel_atom
role	Trust role in SRM. Can only be vendor or owner.	owner	N/A

1. The following example shows the generation of the Owner x509v3 certificate and private key. Use the table above to modify and execute the following command:

```
./SST create-key --role=owner --machine=intel_atom_baytrail --name=ownerE \
--output-dir=./outputE
```

The following two files are placed in the directory `./outputE`:

- ownerE-cert.pem
- ownerE-private.pem

2. If the Owner has a private key, then generate the Owner certificate with the following command:

Note: In the following command `owner-key.pem` is the existing private key.

```
./SST create-key --role=owner --machine=intel_atom_baytrail --name=ownerE --
output-dir=./outputE --priv-key=owner-key.pem
```

The resulting Owner certificate is placed in the directory `./outputE` directory as `ownerE-cert.pem`

5.2 Creating Vendor Public and Private Keys

The Vendor is an authorized entity by the Owner. This section guides you through the required steps of creating a Vendor RSA key pair that is signed by the Owner's private key. The Vendor can provide a private key and use SST to generate a new public cert based on the existing private key.

The following table provides a description of the SST options.

Table 11. Vendor Key SST Commands

Option	Description	Current Value	Default Value
lverbose	Open or close the signing trace. Value can be yes or no	no	no
name	Vendor Name	vendorE	Name of the role
output-dir	The output directory where you can find your private key and CA certificate	./outputE	SST current directory (".")
machine	Target architecture.	intel_atom_baytrail	intel_atom
role	Trust role in SRM. Can only be vendor or owner.	vendor	N/A
issuer	The name of the issuer who delegates to this vendor	ownerE	owner

3. The following example shows the generation of the Vendor x509v3 certificate and private key. Use the table above to modify and execute the following command:

```
./SST create-key --role=vendor --machine=intel_atom_baytrail \
--name=vendorE --output-dir=./outputE --issuer=ownerE
```

This command results in two files in the directory ./outputE: `vendorE-cert.pem` and `vendorE-private.pem`

4. If the Vendor already has a private key, then generate the vendor certificate with following command.

Note: In the following command, `vendor-key.pem` is the existing private key.

```
./SST create-key --role=vendor --machine=intel_atom_baytrail \
--name=vendorE --output-dir=./outputE --priv-key=vendor-key.pem
```

The resulting vendor certificate is placed in directory `./outputE` as `vendorE-cert.pem`

5.3 Signing Bootloader, Kernel, RootFS, RPM Packages

5.3.1 Signing the Bootloader

The SST is used to sign and update the grub.efi bootloader with the appropriate signatures and keys. The update process includes hashing grub.efi and encrypting grub.efi with the Vendor private key. grub.efi is also updated to include the grub.efi signature, Vendor certificate, and Owner certificate.

The following table provides a description of the commands used to sign the bootloader.

Table 12. Bootloader Signing SST Commands

Option	Description	Current Value	Default Value
verbose	Open or close the signing trace. Value can be yes or no.	no	no
machine	The target architecture	intel_atom_baytrail	intel_atom
owner-cert	The root certificate of the trust chain	ownerE-cert.pem	The owner-cert.pem file in the SST current directory
vendor-cert	The device vendor certificate	vendorE-cert.pem	The vendor-cert.pem file in the SST current directory
priv-key	The device vendor private key	vendorE-private.pem	The vendor-private.pem file in the SST current directory

The following example command shows signing the `grub.efi` bootloader. Use the table above to modify and execute the following command:

```
./SST sign-bootloader --machine=intel_atom_baytrail --verbose=no \
--owner-cert=./ownerE-cert.pem --vendor-cert=./vendorE-cert.pem \
--priv-key=./vendorE-private.pem ./grub.efi
```

5.3.2 Signing the Kernel

The SST is used to sign and update the kernel with the appropriate signatures and keys. The update process includes hashing the kernel and encrypting the kernel with the Vendor private key. The kernel also is updated to include the kernel signature and Vendor certificate.

The following table provides a description of the commands used to sign the kernel.

Table 13. Kernel Signing SST Commands

Option	Description	Current Value	Default Value
verbose	Open or close the signing trace. Value can be yes or no.	no	no
machine	The target architecture	intel_atom_baytrail	intel_atom
vendor-cert	The device vendor certificate	vendorE-cert.pem	The vendor-cert.pem file in the SST current directory
priv-key	The device vendor private key	vendorE-private.pem	The vendor-private.pem file in the SST current directory

The following example command shows signing the kernel. Use the table above to modify and execute the following command:

```
./SST sign-kernel --machine=intel_atom_baytrail --verbose=no \
--vendor-cert=./vendorE-cert.pem --priv-key=./vendorE-private.pem \
./ bzImage-initramfs-intel-atom-baytrail.bin
```

5.3.3 Signing RootFS

A signature on RootFS ensures the binary and library in RootFS can be verified successfully after the RootFS is deployed to the Target Device. The kernel and bootloader image in RootFS can also be signed when using SST's sign-all command to sign RootFS.

The following table provides a description of the commands used to sign RootFS.

Table 14. RootFS Signing SST Commands

Option	Description	Current Value	Default Value
verbose	Open or close the signing trace. Value can be yes or no.	no	no
machine	The target architecture	intel_atom_baytrail	intel_atom
vendor-cert	The device vendor certificate	vendorE-cert.pem	The vendor-cert.pem file in the SST current directory
priv-key	The device vendor private key	vendorE-private.pem	The vendor-private.pem file in the SST current directory
Owner-cert	The root certificate of the trust chain	ownerE-cert.pem	The owner-cert.pem file in the SST current directory
Output	The signed RootFS output	./signed-images.tar.bz2	The srm-enabled-images.tar.bz2 file in the current directory
Mode	RootFS type. Value can be tarball or blockfile	tarball	tarball

The following example command shows signing the RootFS tar ball (wrlinux-image-glibc-idp-intel-atom-baytrail.tar.bz2). Use the table above to modify and execute the following command:

```
./SST sign-all --mode=tarball --machine=intel_atom_baytrail --verbose=no \
--vendor-cert=./vendorE-cert.pem --priv-key=./vendorE-private.pem \
--owner-cert=./ownerE-cert.pem --output=./signed-images.tar.bz2 \
./ wrlinux-image-glibc-idp-intel-atom-baytrail.tar.bz2
```

5.3.4 Signing RPM Packages

When the IDP SRM feature is enabled, an RPM package will not install on the Target Device unless the RPM package was signed by the SST.

The following table provides a description of the commands used to sign RPM.

Table 15. RPM Signing SST Commands

Option	Description	Current Value	Default Value
verbose	Open or close the signing trace. Value can be yes or no.	no	no
priv-key	The device vendor private key	vendorE-private.pem	The vendor-private.pem file in the SST current directory
Mode	Sign single RPM or multiple RPM packages. Options are rpm or dir.	rpm	rpm

The following example command shows signing the RPM package (`example.atom.rpm`). Use the table above to modify and execute the following command:

```
./SST sign-rpm --mode=rpm --verbose=no --priv-key=./vendorE-private.pem \
./ example.atom.rpm
```

As an option, you can sign multiple RPM packages with following command:

```
./SST sign-rpm --mode=dir --verbose=no --priv-key=./vendorE-private.pem \
./ rpm-dir
```

The directory `./ rpm-dir` contains RPM packages to sign. After the above command completes, all RPM packages in `rpm-dir` will be signed.

5.3.5 Deploying RootFS Image

Note: The following is a synopsis of the required steps. See the *Wind River Intelligent Device Platform Programmer's Guide XT 2.0* for the complete process of deploying the RootFS image.

1. Enter BIOS setup.
2. Disable secure boot.
3. Delete all secure boot variables.
4. Reboot the system into the USB flash drive.
5. After IDP loads, log in.
6. Transfer the USB flash drive Intelligent Device Platform image to the Target Device HDD. Use the command:

```
tgt=/dev/sda /sbin/reset_media
```

7. Reboot the system.
8. Enter the EFI Shell.
9. Enter the HDD file system.
10. Execute the following command to :
    - Enroll key and certificates into UEFI
    - Lock the BIOS
    - Create the KEK, db, and PK

```
/efi/boot/BOOTIA32.efi
```

6 Configuring UEFI for Secure Boot

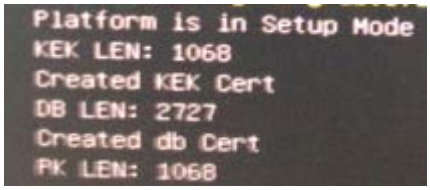
This section provides instructions to enroll the `grub.efi` and kernel keys into the UEFI secure key database to complete the Stage 3 secure boot setup.

6.1 Change the BIOS Settings

1. Insert the USB flash drive into a USB port on the Target Device.
2. Apply power, press the power button, and then immediately press the <Del> key to enter the **Boot Device Menu**.
3. Select **Enter Setup**.
4. Navigate to **Advanced** > **CSM Configuration** > **Video**. Select **UEFI only**.
5. Press <F4>, and select **Yes** to save changes and reboot. Immediately press the <Del> key to enter the **Boot Device Menu**.
6. Select **Enter Setup**.
7. Navigate to the **Advanced** > **CSM Configuration** > **CSM Support**. Select **Disabled**.
8. Press <F4> and select **Yes** to save changes and reboot. Immediately press the <Del> key to enter the **Boot Device Menu.**
9. Select **Enter Setup**.
10. Navigate to **Security** > **Secure Boot Menu** > **Secure Boot**. Select **Disabled**.
11. Navigate to **Security** > **Secure Boot Menu** > **Secure Boot** > **Key Management**. Select **Delete All Secure Boot Variable**, and then select **Yes**.
12. Press <F4> and select **Yes** to save changes and reboot. Immediately press the <Del> key to enter the **Boot Device Menu**.
13. Select **UEFI: Built-in EFI Shell**.

6.2 Enable Secure Boot

1. At the Shell> prompt, type `fs0:` to enter VFAT partition on the USB flash drive.
2. At the `fs0:\` prompt, type `cd EFI\BOOT` to change directories:
3. Use `BOOTIA32.efi` to enroll the keys/certificates into the UEFI and lock down the BIOS. Success is indicated by the following message.

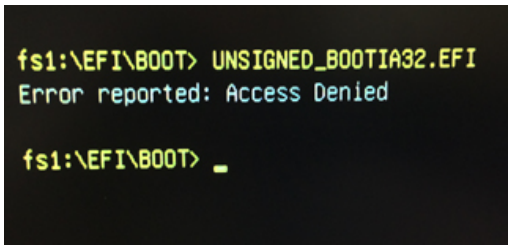


4. Press the <Del> key to enter the **Boot Device Menu**.
5. Select **Enter Setup**.
6. Navigate to **Security** > **Secure Boot menu** > **Secure Boot**. Select Enable.
7. Press the <Del> key.
8. Navigate to **Boot Device Menu** > **UEFI: Built-in EFI Shell**.
9. At the Shell> prompt, type `fs0:` to enter VFAT partition on the USB flash drive.
10. Type `cd EFI\BOOT` to change directories.
11. Run `BOOTIA32.efi`. Verify BOOTIA32.efi can boot the kernel successfully.

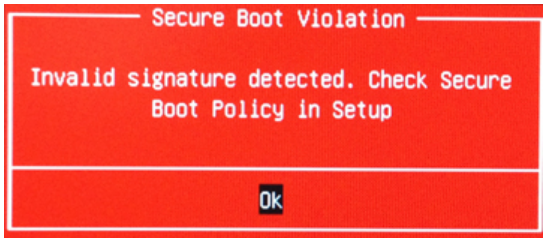
6.3 Test Secure Boot

Use the following steps to make sure that an unsigned `grub.efi` denies kernel boot.

1. Copy an unsigned `grub.efi` to `/EFI/BOOT` in the VFAT partition of your USB flash drive. Name it `UNSIGNED_BOOTIA32.efi`
2. Plug the USB flash drive into the Target Device.
3. Press the <Del> key and select **UEFI: Built-in EFI Shell** option from the **boot device menu**.
4. Run `UNSIGNED_BOOTIA32.efi`. You will receive a warning message similar to the following and you will be denied access. This message indicates that the secure boot policy being enforced:



5. Create an Intelligent Device Platform image with a signature that is different from the signature in the UEFI database. Transfer the image to a USB flash drive.
6. From BIOS menu change the boot device priority to the USB flash drive.
7. Boot from the USB flash drive. You will see the following message:



§