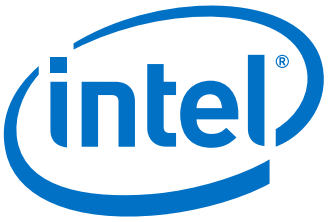
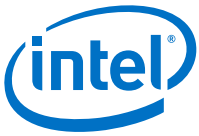


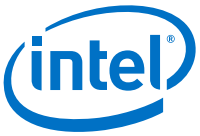
Intel® Quark SoC X1000 Secure Boot

Programmer's Reference Manual (PRM)

March 2014

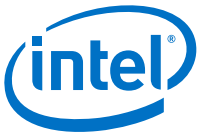
Order Number: 330234-001US





Revision History

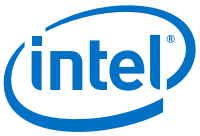
Date	Revision	Description
March 2014	001	First public release of document.



Contents

Figures

Tables



§ §

1.0 Introduction

1.1 About this Manual

The Intel® Quark SoC X1000 is the next generation secure, low-power Intel Architecture (IA) System on a Chip (SoC) for deeply embedded applications. The Intel® Quark SoC X1000 integrates the Intel® Quark Core plus all the required hardware components to run off-the-shelf operating systems and to leverage the vast x86 software ecosystem.

This document describes the Intel® Quark SoC X1000's "secure boot" functionality, which are the mechanisms used to verify the authenticity and integrity of the software executing on the platform prior to that software being loaded and run.

This document contains the following sections:

- Introduction (this section)
- Chapter 2.0, "Security Overview"
- Chapter 3.0, "Secure Boot Overview"
- Chapter 4.0, "Asset Protection"
- Chapter 5.0, "SPI Flash Layout and Asset Signing Tools"
- Chapter 6.0, "Stage 0 (Secure Boot) Execution Details"
- Chapter 7.0, "EDKII Security"
- Chapter 8.0, "Bootloader Security"
- Chapter 9.0, "OS Security"
- Appendix A, "Master Flash Header (MFH) Data Structure"
- Appendix B, "Secure Boot Header Data Structures"
- Appendix C, "Firmware Volume Overview"
- Appendix D, "Sample Flash Layouts"

1.2 Product Documentation

Table 1 lists the documentation supporting this release.

Table 1. Product Documentation (Sheet 1 of 2)

Title	Number
Intel® Quark SoC X1000 Secure Boot Programmer's Reference Manual (this document)	330234
Intel® Quark SoC X1000 Datasheet	329676
Intel® Quark SoC X1000 Linux* Programmer's Reference Manual	330235

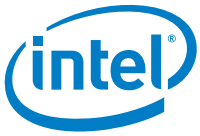


Table 1. Product Documentation (Sheet 2 of 2)

Title	Number
Intel® Quark SoC X1000 Board Support Package (BSP) Build and Software User Guide	329687
Intel® Quark SoC X1000 Software Release Notes	330232
Intel® Quark SoC X1000 UEFI Firmware Writer's Guide	330236

1.3 Terminology

Table 2. Terminology

Term	Description
FV	Firmware Volume
HW RoT	Hardware Root of Trust
IMR	Isolated Memory Region
MFH	Master Flash Header
PCD	Platform Configuration Data
RoT	Root of Trust
SH	Security Header
SMM	System Management Mode
SoC	System on Chip
SVN	Security Version Number
UEFI	Unified Extensible Firmware Interface

1.4 Conventions

The following conventions are used in this manual:

- `Courier` font - code examples, command line entries, API names, parameters, filenames, directory paths, and executables
- **Bold** text - graphical user interface entries and buttons

§ §

2.0 Security Overview

2.1 Assets

In information security, computer security and network security an Asset is any data, device, or other component of the environment that supports information-related activities. Assets generally include hardware (e.g. servers and switches), software (e.g. mission critical applications and support systems) and confidential information.

Source: http://en.wikipedia.org/wiki/Asset_(computer_security)

In the context of Intel® Quark SoC X1000, *Asset Protection* focuses on booting only authenticated software and mechanisms to protect software and/or data from unwarranted access or updates.

Note: The Intel® Quark SoC X1000 does not contain any mechanisms for the protection of confidential information and does not support protection of “Secret Assets”.

2.2 Asset Protection

There are a number of mechanisms provided by the Intel® Quark SoC X1000 to enable the protection of assets. Some of these are provided by default within the SoC design and others can be enabled by software when the system is running.

Chapter 4.0, “Asset Protection” describes the mechanisms provided by the Intel® Quark SoC X1000 for the protection of assets.

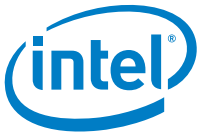
2.3 Secure Boot

One of the most important assets to be protected on a secure platform is the software being run on that platform. The Intel® Quark SoC X1000 provides mechanisms to verify the authenticity and integrity of the software executing on the platform prior to that software being loaded and run. This mode of operation is known as *Secure Boot*.

An overview of *Secure Boot* is provided in Chapter 3.0, “Secure Boot Overview”.

In-depth details of the ROM-based software that provides a *Hardware Root of Trust* are provided in Chapter 6.0, “Stage 0 (Secure Boot) Execution Details”.

Overviews of how EDKII Firmware, Bootloader and OS reference software maintain a *Chain of Trust* are contained in chapters Chapter 7.0, “EDKII Security”, Chapter 8.0, “Bootloader Security” and Chapter 9.0, “OS Security”. These chapters reference the relevant programmer’s reference manuals where further details can be obtained.



2.4 Asset Signing

Some assets, notably software applications that are authenticated before being allowed to run and associated key material, require post-processing as part of the build flow in order to add their signature prior to use on an Intel® Quark SoC X1000 based platform.

Chapter 5.0, "SPI Flash Layout and Asset Signing Tools" describes the post-processing reference tools provided by Intel to carry out this asset signing.

2.5 Implementation Note

Note: Whilst the Intel® Quark SoC X1000 ROM software run from reset **requires** the first software application that is loaded, prepared and signed in accordance with the requirements set out in this document, it is open to the customer to decide upon an appropriate mechanism for authenticating later stage applications. Chapter 7.0, "EDKII Security", Chapter 8.0, "Bootloader Security", and Chapter 9.0, "OS Security" of this document contain high-level details and links to the relevant documents detailing the implementation details used in reference Intel® Quark SoC X1000 based solutions provided by Intel.

§ §

3.0 Secure Boot Overview

3.1 Introduction

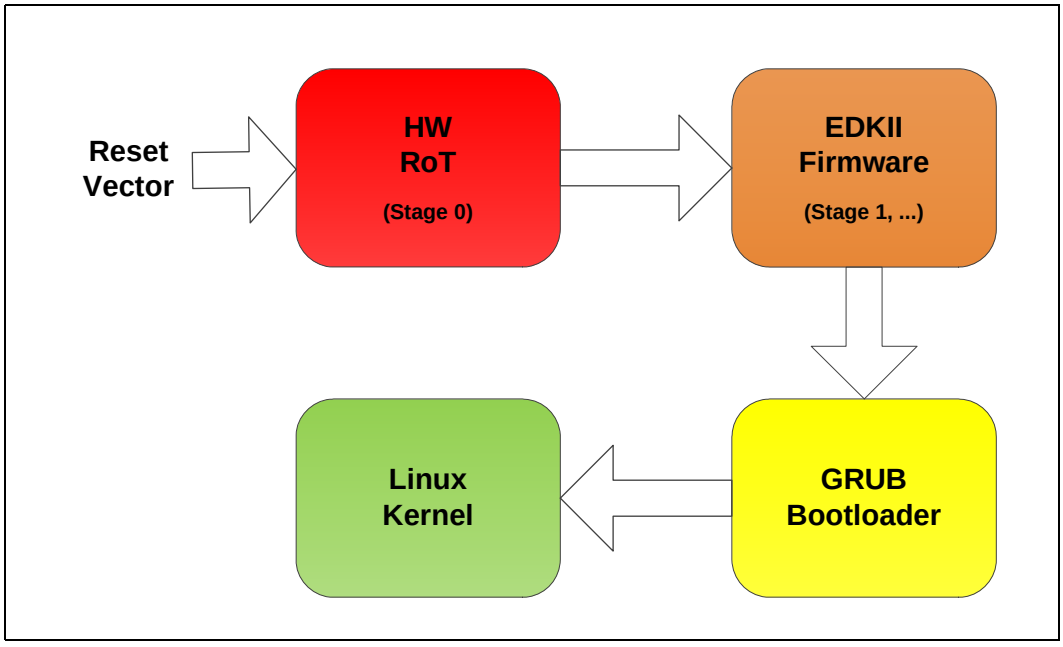
For secure embedded applications, it is necessary to verify the authenticity and integrity of software executing on the platform prior to that software being loaded and run. This mode of operation is known as *Secure Boot*.

The boot flow starts with the authentication of the first stage boot loader executed during the boot sequence prior to it being executed and is repeated all the way through the boot sequence. The authentication process is iterative – execution stage *N* authenticates execution stage *N+1* which in turn authenticates execution stage *N+2* and so on.

To begin this process, it is necessary to provide a known and secure point from which the iterative authentication process can begin (this is referred to in this document as *Stage 0*). This is known as establishing a *root of trust*. The Intel® Quark SoC X1000 establishes a *Hardware Root of Trust* by executing software contained within an on-die ROM from the point of reset. This software is responsible for authenticating a Stage 1 application and passing control to it.

A typical boot flow showing the chain of trust is provided in Figure 1.

Figure 1. Chain of Trust in the Boot Flow



3.2 Asset Signing and Authentication Process

An industry standard mechanism is used for the digital signing of assets for Intel® Quark SoC X1000 platforms. There are two main steps used each in the signing and authentication processes. For both processes, the first step involves calculating a unique *hash* value of the asset using the *SHA 256* algorithm. For the signing process, the second step involves creating a *signature* of that hash value using RSA encryption. For authentication, the second step involves RSA decryption of the signature for validation by comparison with the calculated hash value. The RSA algorithm used is RSA 2048 with the PKCS1-PSS padding scheme.

3.2.1 Signing an Asset

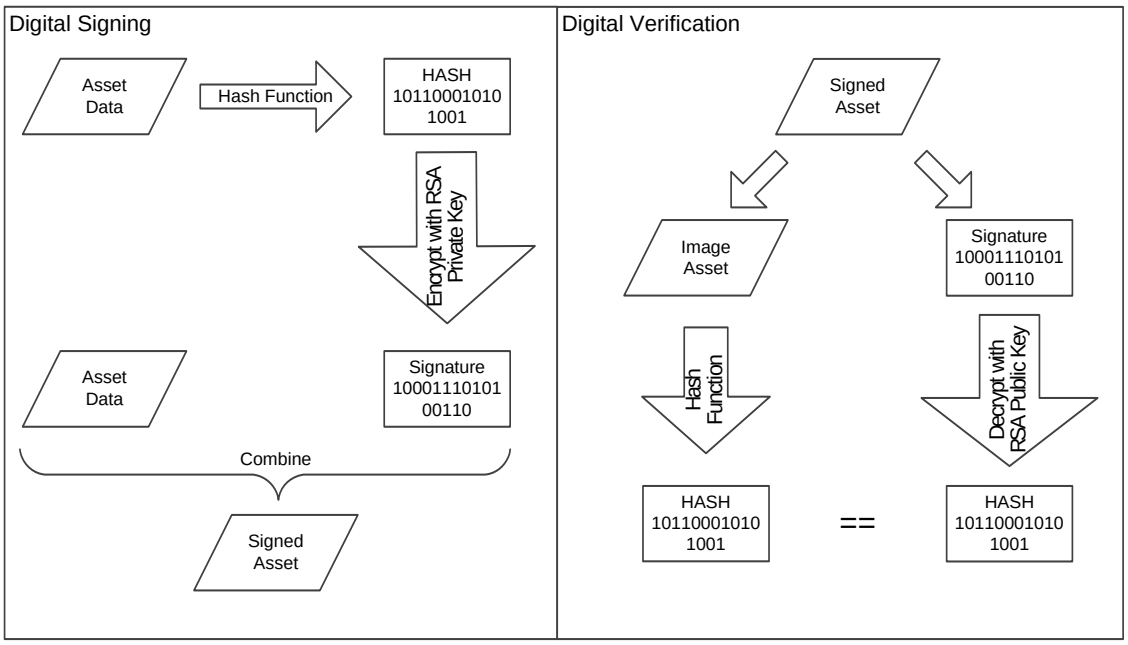
In the first stage of signing an asset, a SHA-256 hash value is calculated on the asset. An encrypted copy of this hash value is then created using a Private Key and the RSA algorithm. This is referred to as the *signature* of the asset. The signature is then added to the asset (see Chapter 5.0, "SPI Flash Layout and Asset Signing Tools" for Intel® Quark SoC X1000).

3.2.2 Validating an Asset Signature

The integrity of the asset can be verified by again calculating a SHA-256 hash value of the asset. The signature that was attached to the asset is decrypted using the Public Key associated with the key used to create the signature. If the decrypted signature matches the calculated asset hash value, the asset is valid.

Figure 2 shows the digital signing/verification mechanism for an asset.

Figure 2. Digital Signing / Validation



3.2.3 Additional Authentication Steps

As part of the packaging of an asset together with its signature, a header is created describing such details as the size of the asset, the algorithms used to hash and sign the asset and key sizes. The header also contains a mechanism to prevent roll-back to valid assets that may not be loaded because of, for example, a security vulnerability. This mechanism is called Security Version Numbering, details of which are described later in Section 4.6, “Rollback Protection - Security Version Numbers” on page 23.

In addition to checking the signature of an asset, the authentication software checks the contents of the header and does not allow an asset that does not match its requirements to be loaded.

3.3 RSA Keys

The signing algorithm used for the Secure Boot flow is RSA, with a key size of 2048 bits. The RSA algorithm uses a Private/Public key pair for the encryption/decryption processes. The private key is used to encrypt a signature and should be known only to the entity creating the signature. The public key is used to decrypt the signature and can be generally distributed and used. Because the public key is associated directly with the (secret) private key, anyone decrypting a signature with the public key can be confident that it was encrypted by the appropriate signing party if the correct signature is decrypted. Management of the Private key is paramount, as only the appropriate signing party should have access to this key.

3.4 Key Handling During Stage 0

The Hardware Root of Trust (HW RoT) on Intel® Quark SoC X1000 is made up of two components:

- The software executed from the on-die ROM.
- The public half of the first RSA key (stored on-die) used in the Boot flow validation process.

The on-die key, referred to as the *Device Key,* is programmed by the factory at Intel. If this key were to be used directly in the verification of Stage 1 applications, then either Intel or Original Equipment Manufacturers would have to sign all Stage 1 applications on behalf of developers, which is not a feasible situation.

As a result, the Secure Boot code executing in Stage 0 is broken into two stages:

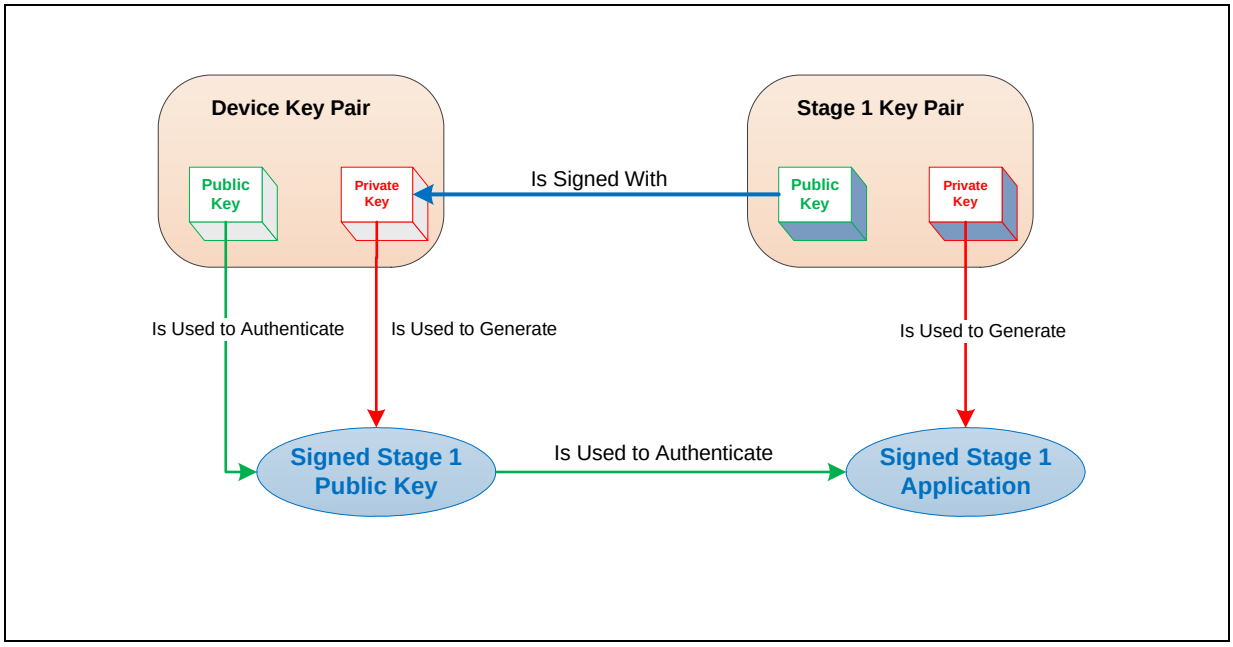
1. The authentication of an RSA Public key stored in SPI Flash.
2. The use of this key to then validate the Stage 1 application.

This second key is referred to as the *Stage 1 Key*. The private half of the Stage 1 Key is held and managed by the developer and they use this to sign all of their Stage 1 applications. The public half of this key is submitted to Intel for signing with the private half of the Device Key Pair (held and managed by Intel).

Note: For convenience during development, the software release includes a default Private Key `key.pem` file. During development, all assets are signed with the default key that is stored in the `config` directory. The default key **cannot** be used in a production system; it is not secure due to its inclusion in the release package. Contact your Intel representative for details.

Figure 3 illustrates the usage of both the Device Key and Stage 1 keys.

Figure 3. Stage 0 Key Usage



3.5 Storage of Stage 1 Public Key and Stage 1 Applications

Since the Stage 0 software operates from on-die ROM, it must know how to find the Signed Stage 1 Public Key and Signed Stage 1 Applications in order to authenticate and launch a Stage 1 Application.

The Stage 1 Public Key is stored in a structure known as the *Key Module* and is stored at a fixed location in the Legacy SPI Flash device. It is possible to have multiple Stage 1 applications (for redundancy, fault handling, and so on), all of which are also stored in the Legacy SPI Flash device and a mechanism for locating each of those images is described below.

3.5.1 Key Module

The Key Module contains a signed copy of the public half of the Stage 1 Key to be used during the authentication of Stage 1 applications. The location of the Key Module in Legacy SPI Flash is fixed by the Stage 0 code implementation. Further details of where to store it in SPI Flash and the usage of supplied tools to place the module in a Flash image are contained in Chapter 5.0, “SPI Flash Layout and Asset Signing Tools” for Intel® Quark SoC X1000.

Note: If a Secure Intel® Quark SoC X1000 attempts to boot from an SPI Flash device that does not contain a valid Key Module at the required location, the system fails to boot and instead enters an infinite idle loop.

3.5.2 Stage 1 Applications

3.5.2.1 Master Flash Header

The Intel® Quark SoC X1000 supports the ability to have multiple Stage 1 Applications on an SPI Flash device (for redundancy, recovery, and so on) and does not require those applications to be located at fixed addresses in the SPI Flash device. This affords flexibility to developers but in turn, requires a mechanism for communicating the location of the applications to the Stage 0 code. This is done using a structure known as a *Master Flash Header*, which maps out the content of the SPI Flash device. The Master Flash Header contains information on many types of application and data stores within the Flash device, not just Stage 1 applications, but is used by the Stage 0 code specifically to locate Signed Stage 1 applications.

3.5.2.2 Stage 1 Applications

Since there can be multiple Stage 1 applications stored in the Flash device, a priority scheme must be applied to determine which one should be executed. The Stage 0 code follows the list of assets in the Master Flash Header, attempting to authenticate each Stage 1 application as they are found. As soon as the Stage 0 code succeeds in authenticating a Stage 1 application, control passes to that application and the remaining Stage 1 applications are not inspected.

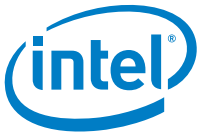
The location of the Master Flash Header in Legacy SPI Flash is fixed by the Stage 0 code implementation. Further details of the structure of the header, how to generate a header, where to store it in SPI Flash and the usage of reference tools to place the module and all Stage 1 applications in a Flash image are contained in Chapter 5.0, “SPI Flash Layout and Asset Signing Tools” for Intel® Quark SoC X1000.

3.5.2.3 Fixed Location Recovery Application

Should all available Stage 1 applications fail to be authenticated, or indeed should the Master Flash Header get corrupted in some way such that it is not possible for the Stage 0 code to locate the applications, the Intel® Quark SoC X1000 provides a mechanism whereby a developer can place a Stage 1 application at a known fixed address. The Stage 0 code, if it fails to authenticate any Stage 1 application via the Master Flash Header will finally attempt to locate and validate a Stage 1 application at a fixed location. This is referred to as the *Fixed Location Recovery Application* and while it could contain any Stage 1 application, it is anticipated that this application is typically used as a final effort to get the system into a Recovery Mode.

The location of this application in Legacy SPI Flash is fixed by the Stage 0 code implementation. Further details of where to store it in SPI Flash and the usage of supplied tools to place the signed application in a Flash image are contained in Chapter 5.0, “SPI Flash Layout and Asset Signing Tools” for Intel® Quark SoC X1000.

Note: If a secure Intel® Quark SoC X1000 cannot authenticate any Stage 1 applications, including the Fixed Location Recovery Application, the system fails to boot and instead enters an infinite idle loop.



3.6 Stage 1 Execution Environment

The Intel® Quark SoC X1000 contains 512 KBytes of on-die embedded SRAM, referred to as eSRAM. This SRAM can be used for code and/or data storage and is available from system startup.

As part of the boot flow, the Stage 0 software configures the eSRAM, transfers the Stage 1 Application from SPI Flash to the eSRAM, authenticates it and then passes control to that application, running from eSRAM.

The configuration of eSRAM and the addresses used within the eSRAM for storage and execution of the Stage 1 Application are fixed by the Stage 0 code implementation.

This means that there is a requirement for Stage 1 Applications to be built to be run from a specific address range in the memory map. This requirement feeds into the configuration of build tools used to generate Stage 1 applications. Further details of the relevant requirements and settings are described in the Intel® Quark SoC X1000 UEFI Firmware Writer's Guide.

3.7 Memory Protection

The Intel® Quark SoC X1000 contains a mechanism for protecting memory from unwanted access by system agents (for example, some I/O peripheral attempting to write to a portion of memory that contains execution code and that should only be accessible to the core). This mechanism is referred to as Isolated Memory Regions (IMRs). Eight Isolated Memory Regions are supported and are used as part of the Secure Boot flow to ensure software that has been authenticated cannot be modified after it has been authenticated by any system agent other than the core. Further details of IMR settings and usage during Secure Boot are contained in Chapter 4.0, "Asset Protection."

3.8 Stage 0 Software Execution Overview/Flow

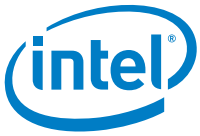
The following is a high-level overview of the flow of execution of the Stage 0 code:

1. The system is taken out of reset.
2. Embedded SRAM is configured.
3. An Isolated Memory Region is configured to protect the eSRAM address range.
4. The Key Module is copied to eSRAM, authenticated and the Stage 1 Public Key extracted.
   a. If authentication fails, the system enters an infinite idle loop.
5. A check is done to see if a Master Flash Header exists.
   a. If it does exist, flow continues from step 6.
   b. If it does not exist, flow continues from step 9.
6. The Master Flash Header is searched for the first Signed Stage 1 application.
7. The Signed Stage 1 application is copied to eSRAM.
8. An attempt is made to authenticate the Stage 1 application.
   a. If authentication is successful, execution is passed to that application and Stage 0 is complete.
   b. If authentication fails, the Master Flash Header is searched for the next Signed Stage 1 application. If an application is found, flow continues from step 7.
   If no application is found, flow continues from step 9.
9. The Fixed Location Recovery Application is copied to eSRAM.

10. An attempt is made to authenticate the application.
    a. If authentication is successful, execution is passed to that application and Stage 0 is complete.
    b. If authentication fails, the system enters an infinite idle loop.

Further details of the Stage 0 software execution are contained in Chapter 6.0, “Stage 0 (Secure Boot) Execution Details”.

§ §



4.0 Asset Protection

As mentioned in the overview, in information security, computer security and network security is an *Asset* is any data, device, or other component of the environment that supports information-related activities. Assets generally include hardware (for example, servers and switches), software (for example, mission-critical applications and support systems) and confidential information.

This chapter provides an overview of the mechanisms provided by the Intel® Quark SoC X1000 to protect assets in software applications throughout the boot flow as well as describing the specific steps taken by Stage 0 (ROM) software to protect Stage 1 application software.

4.1 Reset Handling

The design of the Intel® Quark SoC X1000 ensures that upon receipt of resets and S3 exit triggers, the ROM Based Stage 0 Software is run in all cases.

4.2 ROM Based Software - Hardware Root of Trust

The root of trust in the Intel® Quark SoC X1000 is a hardware root of trust by virtue of the fact that the first code executed after reset is ROM based. This software (using the hardware-based *Device Key* as the first part of the flow) authenticates the next piece of software before allowing it to run. Details of this ROM-based software are described in Chapter 3.0, “Secure Boot Overview” and Chapter 6.0, “Stage 0 (Secure Boot) Execution Details”.

4.3 Hardware Based Authentication Key

Software applications are signed and authenticated using RSA public/private key pairs. The very first key used in the boot flow is stored in the Intel® Quark SoC X1000 hardware. Details of this key and its use are described in Chapter 3.0, “Secure Boot Overview” and Chapter 6.0, “Stage 0 (Secure Boot) Execution Details”.

4.4 Cache Settings

During early reset handling, the Intel® Quark Core cache is invalidated and disabled. To minimize boot time, the regions of memory covering the Secure Boot ROM and eSRAM (as configured by the Stage 0 software) are cached. It is the responsibility of Stage 1 applications to reconfigure these cache settings as appropriate to the particular application.

4.5 Isolated Memory Regions (IMR)

The Intel® Quark SoC X1000 supports a mechanism to protect memory regions from unwarranted access by agents in the system that should not have access to that memory. For example, resource tables used to convey information from BIOS to an OS

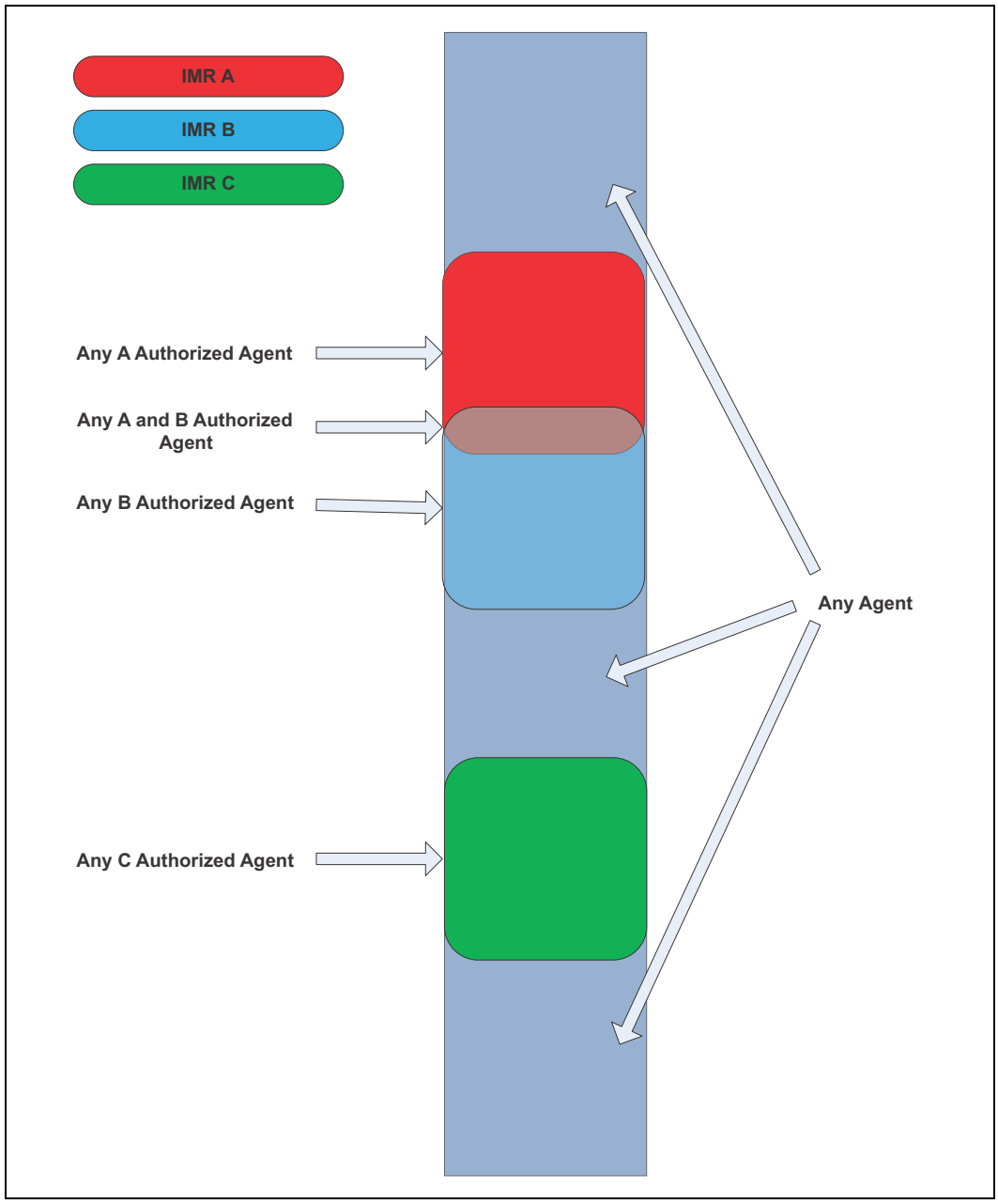
should not be capable of being directly modified by the hardware of a peripheral device (for example, by DMA transfer). This protection mechanism is referred to as *Isolated Memory Regions*.

The Intel® Quark SoC X1000 supports eight general purpose IMRs. These IMRs are configured during the boot flow and are used to protect both software applications that are being authenticated (to ensure software that is authenticated is unchanged, for example, by a peripheral changing content after the validation, when it is run). IMRs are configured to indicate which system agents (the CPU, peripheral devices, Remote Management Unit and so on) are allowed to read from the memory region or write to that memory region. The memory region is defined by a start and end address pair and has a 1 KB resolution.

In a configuration where two IMRs overlap, only agents that are enabled by both IMRs are allowed access to the overlapping address range.

Figure 4 illustrates the behavior of three IMRs showing regions of memory with no IMR restrictions, non-overlapping IMR (IMR C) and overlapping IMRs (A & B).

Figure 4. Example IMR Zone Accesses



4.5.1 IMR Usage During Boot Flow

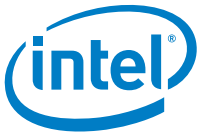
Software applications and critical data used during the boot flow fall into two categories, data that is protected for a relevant portion of the boot flow and data that must be protected during the life of the system.

Table 3. IMR Usage During Boot Flow

IMR	ROM	Stage 1	Stage 2	Grub	Linux* Boot	Linux* Run-time
0		Compressed EDKII stage 2 Uncompressed EDKII stage 2 Boot time services Grub Image Stack/Data area	Compressed EDKII stage 2 Uncompressed EDKII stage 2 Boot time services Grub Image Stack/Data area	Compressed EDKII stage 2 Uncompressed EDKII stage 2 Boot time services Grub Image Stack/Data area	Compressed EDKII stage 2 Uncompressed EDKII stage 2 Boot time services Grub Image Stack/Data area	Uncompressed Kernel Read only & initialized data section
1		AP Startup vector	AP Startup vector	Boot Params	Boot Params	
2		Shadowed Quark SoC Remote Management Unit main execution binary				
3		Low SMRAM	Low SMRAM		Entire Memory (4G)	
4		eSRAM protection during late stage 1 and early stage 2 phases (EDKII stage 1 post memory initialization)				
5			Legacy S3 memory	Legacy S3 memory	Legacy S3 memory	Legacy S3 memory
6			ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory	ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory	ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory	ACPI NVS Runtime Code Runtime Data Reserved memory ACPI Reclaim memory
7	eSRAM protection during ROM phase (EDKII stage 1)	eSRAM protection during early stage 1 phase (EDKII stage 1 pre memory initialization)		Compressed Kernel OS Image	Compressed Kernel OS Image	

4.5.2 IMR Violation Behavior

In addition to configuring the settings for individual Isolated Memory Regions, the IMR hardware must be configured to enable the reporting of IMR Violations (by setting the `EnableIMRInt` bit in the `BIMRVCTL` register). Setting this bit results in the Intel® Quark SoC X1000 being reset upon detection of an IMR violation. Stage 0 ROM software sets this configuration bit and it is recommended that later stage software does not change this setting.



4.5.3 SMRAM and HMBOUND – Special IMRs

As per typical Intel Architecture design, the Intel® Quark SoC X1000 defines a register named HMBOUND that enables firmware to define the amount of DRAM currently on the system through the specification of the top of that memory region. Any attempt by an agent in the system to access a memory address above the value set by HMBOUND results in an HMBOUND-IMR Violation. System behavior in the event of HMBOUND violations shall be identical to standard IMR violations.

Also per typical Intel Architecture design, the Intel® Quark SoC X1000 defines an operating mode called System Management Mode (SMM) with which is associated an area of memory that is only available for use when in SMM and is called SMRAM. Any attempt by a system agent to access this memory or any attempt by the Core CPU to access this memory when not in SMM results in an SMRAM-IMR Violation. System behavior in the event of SMRAM violations shall be identical to standard IMR violations.

Further details on configuring HMBOUND can be found in the Intel® Quark SoC X1000 Datasheet.

Further details on SMM and configuring SMRAM can be found in the Intel® Quark SoC X1000 UEFI Firmware Writer's Guide.

4.5.4 IMR Locking

As mentioned in Section 4.5.1, "IMR Usage During Boot Flow" on page 22, IMRs can be used for a limited period of time (*Temporary IMR*) or for the life of a system (*Permanent IMR*). Once a Permanent IMR has been configured, it should be locked to ensure protection is maintained for the life of the system.

Temporary IMRs should not be locked during usage to allow the IMR to be reused. However, to protect against potential attacks whereby an IMR is used to block legitimate access to memory, before the OS boot software hands off to the OS runtime, all unused IMRs *must* be disabled and locked.

Further details on IMRs, their configuration and a description and list of system agents that can be impacted by IMR setup can be found in the Intel® Quark SoC X1000 Datasheet.

4.6 Rollback Protection - Security Version Numbers

During the life of a product, it is possible for a product vendor, when releasing a new software version, to want to prevent systems rolling back to a previous release. For example, when a security vulnerability is identified in the earlier version. The earlier version of software is signed with valid keys and passes all of the validation tests on the headers associated with that software. Therefore, a mechanism is required to indicate to systems that they should not run "older" versions of software. This mechanism is referred to as *Security Version Numbers (SVNs)*.

The Intel® Quark SoC X1000 supports the concept of having an array of Security Version Numbers, with each number being associated with a specific asset. The Secure Boot ROM Code claims the first three SVNs, used for:

- SVN0: Key Module
- SVN1: Stage 1 Software Applications
- SVN2: Fixed Location Recovery Application

4.6.1 SVN Storage

The SVN Array is stored at a fixed location in the Legacy SPI Flash Device, physical address FFFD0000h in the Intel® Quark SoC X1000 memory map. This allows for a maximum of 32 KB of storage space up to the next fixed location in the memory map (which is the Key Module).

Note: The address range selected for the SVN Array should fall within the Protected Boot Block area of the SPI Flash Device used. It is recommended that the Protected Boot Block mechanism be enabled by Bootloader software to ensure that this protection is applied to the SVN Array.

4.6.2 SVN Usage

Each SVN is a 32-bit integer. When validating a signed asset, the validation software compares the SVN stored at the relevant offset in the SVN Array with the SVN stored in the Secure Boot Header of the asset being authenticated. If the SVNs match or the SVN attached to the asset is higher than that in the array, the asset validation can proceed. If the SVN from the SVN array is lower than the asset SVN, validation is deemed to have failed.

Note: Updating the SVN is the responsibility of Update and Recovery mechanisms in the Bootloader and/or OS. When an SVN of an asset that is being updated is higher than the current SVN that is detected, the array should be updated to reflect this higher level SVN.

4.7 Interrupts

As part of the Secure Boot ROM software execution, interrupts are disabled during the validation of Stage 1 applications. It is recommended during each phase of the boot flow that only interrupts that are required to support the boot flow are enabled and that they remain enabled only for as long as they are required.

4.8 SPI Flash Device - Protected Boot Block

SPI Flash devices contain regions that support the concept of Boot Block Protection, whereby the device can be configured not to allow any erase or write attempts until a reset occurs. This protection can mitigate both against accidental attempts to update the device as well as malicious attempts to change contents.

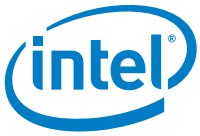
The specific mechanism to enable the protection, as well as the number and size of blocks, is flash device dependent and outside the scope of this document.

It is recommended that:

- Boot Block Protection be enabled by Stage 1 Bootloader applications.
- At a minimum, the following resources be contained within protected boot blocks:
  - Key Module
  - SVA Area
  - Fixed Location Recovery Application

4.9 SPI Flash - Write Protect Mode

Intel architecture devices support a mechanism that allows access to SPI Flash devices attached to the legacy port to be put into *Write Protect Mode*. In this mode, attempts to erase or write to any sector of the Flash device are disabled by hardware. This



protection can mitigate both against mistaken attempts to update the device, as well as malicious attempts to change contents. The protection is enabled by UEFI Firmware once it determines that it has completed any updates it may make to the Flash device.

Details of the steps to achieve this protection are contained in Chapter 8.0, “Bootloader Security”.

§ §

5.0 SPI Flash Layout and Asset Signing Tools

5.1 Introduction

Intel® Quark SoC based platforms contain legacy SPI flash which is used to boot the platform. In Non-Secure Boot mode, SPI flash contains reset code and boot images. In Secure Boot mode, SPI flash contains signed boot images.

The process of integrating multiple modules together into an SoC-compatible SPI Flash image can be managed using the SPI Flash Layout tool.

The Asset Signing Toolset provides a method for the SPI Flash Layout tool to sign modules. It can also be used as a standalone tool to sign modules individually.

The SPI Flash layout and Asset Signing tools provide a simple and flexible method to create a Secure Boot and Non-Secure Boot SoC-compatible SPI Flash image.

5.2 SPI Flash Layout Tool

5.2.1 Overview

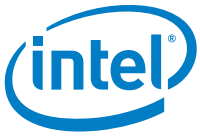
The SPI Flash Layout tool enables a user to create a custom flash image with the ability to:

- Select modules and state the address at which they will reside in the SPI flash image.
- Specify if a asset(s) is to be signed by the asset signing tool.
- Specify if the asset(s) is to be wrapped with a firmware volume.
- Specify if a Master Flash Header (MFH) is to be included in the flash image and also the ability to specify which asset(s) will be detailed in the Master Flash Header.

5.2.2 Note on Fixed Address Usage

The SPI Flash device occupies the very top of the address space on each Intel® Quark SoC X1000 based platform. However, when specifying an address in the layout tool this address is relative to the start of the flash device (and can also be considered an offset into the Flash device). This means that any “fixed” address from a platform view will occupy a different offset into different sized Flash devices.

All “fixed” addresses provided in this document are calculated on the basis of an 8 MB Flash device being used. Those addresses must be adjusted in any case where a different sized Flash device is used.



5.2.3 Pre-Requisites

Before using the SPI Flash Layout tool, the steps in the Intel® Quark SoC X1000 Board Support Package (BSP) Build and Software User Guide must have been completed. See the section “Creating a flash image” for details.

5.2.4 Image Generation

To generate a SPI Flash image, first a valid `layout.conf` must be provided in the root directory of the SPI Flash Layout Tool. The next step is to run the `make` command in the same directory. The resulting SPI Flash Image is a binary file named `Flash.bin`. This generated SPI Flash Image will be laid out in accordance with the supplied `layout.conf` file.

Note: If the user wishes to change the `layout.conf` file and regenerate an image, they must first run a `make clean` before running `make` again.

5.2.5 The Layout Configuration File

The user designs the flash layout using a `layout.conf` configuration file, which contains the following descriptor blocks:

- Main Descriptor Block
- Standard Asset Descriptor Block
- Master Flash Header Asset Descriptor Block
- Debug Dump Asset Descriptor Block

5.2.5.1 Main Descriptor Block

This block of the `layout.conf` file defines the size of the SPI flash image (`Flash.bin`).

Example 1. Main Descriptor Block

```
[main]
size=8388608
type=global
```

5.2.5.1.1 Size

Size of the SPI flash in bytes. Only values of 4M or 8M are supported.

5.2.5.1.2 Type

Type field `global` indicates that this descriptor block definition applies to the entire `layout.conf` file.

5.2.5.2 Standard Asset Descriptor Block

The following is an example of a standard layout descriptor block. Each field is described below the example.

Example 2. Standard Asset Descriptor Block

```
[boot_stage1_image1]
address=0x6C0000
item_file=modules/boot_stage1.fv
fvwrap=no
guid=none
sign=yes
boot_index=0
type=mfh.host_fw_stage1_signed
svn_index=1
```

5.2.5.2.1 Asset Name

The first field in the `[boot_stage1_image1]` example is the asset descriptor name. This is an underscore separated text description of the asset descriptor block and is surrounded by square brackets. This name *does not* have to be unique and its sole purpose is user readability and organization of the `layout.conf` file.

5.2.5.2.2 Address

The `address` field takes a hexadecimal value that specifies where the asset file is placed in the generated SPI Flash image. This address is relative to the start of the SPI Flash.

5.2.5.2.3 Fvwrap

The `fvwrap` field specifies if an asset is to be wrapped in a Firmware Volume. This field has two option values ‘yes’ or ‘no’.

The SPI FLash Layout tool uses the Firmware Volume Tools from the EDK2 Basetools package to provide firmware volume wrapping. For information on how to achieve this manually, refer to the Using FV Tools section in Appendix C, “Firmware Volume Overview”.

An overview of the Firmware Volume format can be found in Appendix C, “Firmware Volume Overview”.

5.2.5.2.4 Guid

The `guid` field specifies the firmware volume name when the `fvwrap` field value is set to ‘yes’.

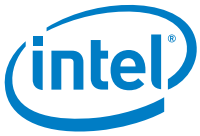
A `guid` is a Globally Unique Identifier. A 128-bit value used to name an entity uniquely. In this case, it is used as the FV Name.

The `guid` field value is specified in the following format:
‘XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX’ or ‘none’

5.2.5.2.5 Sign

The `sign` field specifies if an asset is to be signed or not. This field has two option values ‘yes’ or ‘no’.

Note: When the layout tool finds an asset that needs to be signed, it calls the Asset Signing Toolset to sign the asset before adding it to the SPI Flash image. An in-depth description of the Asset Signing Toolset can be found in Section 5.3.



5.2.5.2.6 Boot_Index

The `boot_index` specifies the Master Flash Header boot index value for a bootable asset. This is an integer value. For more information on this value, refer to the Boot Index (0..n) section in Appendix A, "Master Flash Header (MFH) Data Structure".

If an asset does not require a boot index, its value can be set to 'none'.

5.2.5.2.7 Type

The `type` field, indicates to the SPI Flash Layout tool the type of asset that is being described. Upon Image generation the asset is then tagged in the MFH with this type value. MFH types can be found in Table 9 in Appendix A, "Master Flash Header (MFH) Data Structure". In the case of an asset that is not to be included in the MFH, any value can be provided.

5.2.5.2.8 svn_index

The `svn_index` field has an integer value from 0-15 or 'none'.

A Security Version Number must be supplied for a signed asset, for more information on Security Version Numbers, refer to Security Version Number Indexing in Appendix B, "Secure Boot Header Data Structures".

Note: The SVN Index must align to the correct MFH type. Refer to Table 10 in Appendix A, "Master Flash Header (MFH) Data Structure" for SVN Index allocation.

Note: A SVN area module and a Key Module asset must be present in the SPI Flash image for Secure Boot to work. The SVN module *must be* placed at address 0x7D0000. The Key Module *must be* placed at address 0x7D8000 for an 8 MB part.

5.2.5.3 Master Flash Header Asset Descriptor Block

The Master Flash Header serves as a road map for the contents of SPI flash. It contains the location and size of each element in the flash. A full in-depth description of the Master Flash Header can be found in Appendix A, "Master Flash Header (MFH) Data Structure".

The following is an example of a Master Flash Header asset descriptor block. Each field is described following the example.

Note: To include a Master Flash Header in the SPI Flash image, first a MFH block must be included in the `layout.conf` file.

Example 3. Master Flash Header Asset Descriptor Block

```
[MFH]
version=0x1
flags=0x0
address=0x708000
type=mfh
```

5.2.5.3.1 Version

The `version` field value is a 32 bit hexadecimal number. It specifies which revision of the Master Flash Header is present. The current version is 0x00000001.

5.2.5.3.2 Flags

The `flags` field value is a 32 bit hexadecimal number that can be used in the Master Flash Header as flags. Currently, this value is reserved and unused.

5.2.5.3.3 Address

The `address` field takes a hexadecimal value that specifies where the Master Flash Header is placed in the generated SPI Flash image.

Note: The Master Flash header *must be* placed at address 0x708000 for an 8 MB part.

5.2.5.3.4 Type

The `type` field indicates to the SPI Flash Layout tool what type of asset is being specified. In this case, this value must be 'mfh', informing the tool that we want an MFH to be created and placed at the provided address.

Note: To include an asset in the Master Flash Header, each asset descriptor block must include an MFH type value.

5.2.5.4 Debug Dump Asset Descriptor Block

This non-standard asset descriptor block is used to dump or place the current `layout.conf` file into the generated SPI Flash image. This is a useful debug feature and may be extended in the future.

The following is an example of a Layout Conf Dump Asset descriptor block. Each field is described following the example.

Example 4. Debug Dump Asset Descriptor Block

```
[LAYOUT.CONF_DUMP]
address=0x708200
type=mfh.build_information
meta=layout
```

5.2.5.4.1 Address

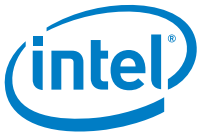
The `address` field takes a hexadecimal value that specifies where the `Layout.conf` file dump is placed in the generated SPI Flash image. This address is relative to the start of SPI Flash.

5.2.5.4.2 Type

The `type` field indicates to the SPI Flash Layout tool what type of asset is being specified. In this case the type value of `mfh.build_information` is required.

5.2.5.4.3 Meta

The `meta` field must contain the value `layout`. No other values are supported.



5.3 Asset Signing Toolset

5.3.1 Overview

The Asset Signing Toolset enables a user to generate the Secure Boot Header for an asset and prepend it to that asset.

For information on the Secure Boot Header, refer to Appendix B, “Secure Boot Header Data Structures”.

The Asset Signing Toolset performs the following functions:

- Prepends the Security Header (64 Bytes) portion of the Secure Boot Header.
- Applies a padding from the Security Header to a user specified offset.
- Generates a SHA256 hash of the user provided binary.
- The RSA 2048 signature is generated using the SHA256 hash and the user provided Private Key.

  Note: For convenience during development, the software release includes a default Private Key `key.pem` file. During development, all assets are signed with the default key that is stored in the `config` directory. The default key **cannot** be used in a production system; it is not secure due to its inclusion in the release package. Contact your Intel representative for details.
- The RSA Public Key and the RSA Signature portion of the Secure Boot Header is then added to the resulting signed binary.

The toolset provides two ways to sign files:

1. Prefixing the file with the security header. This produces a `.signed` file which is a combination of the header and the original file.
2. Producing a “standalone” signature file, `.csbh` file, next to the original file.

In the current release:

- assets in the SPI flash support `.signed` files only.
- assets on SD/USB support `.csbh` files only.

5.3.2 Pre-Requisites

Before using the Asset Signing Toolset, the steps in the Intel® Quark SoC X1000 Board Support Package (BSP) Build and Software User Guide, in the section “Signing Files” must have been completed.

5.3.3 Using the Asset Signing Toolset

The asset signing toolset has the following command line options.

5.3.3.1 -i <input file>

The binary file to be signed.

5.3.3.2 -o <output file> (optional)

The output file. If no filename is provided, the tool will use the input filename and append `.signed` (default) or `.csbh` (if `-c` option is used).

5.3.3.3 -b <body offset in hexadecimal> (optional)

This is the hexadecimal offset between the Security Header and the asset body. A minimum value of 588 bytes (0x24C) is required.

Warning: The Body Offset for Stage1 modules must match the expected Stage1 header size. This is to ensure the Stage1 entry point builds to the correct address.

5.3.3.4 -s <svn>

The Security Version Number (SVN) is a field used to prevent roll-back of software images to previous versions that may have a security vulnerability. This number is compared to a stored value on the platform and the image will only load if the value is equal to or greater than that stored on the platform.

5.3.3.5 -x <svn index>

You must select the appropriate SVN index for your asset based on the allocation described in Section B.4, “Security Version Number Indexing” on page 51.

5.3.3.6 -k <key file>

The Asset Signing tool uses an industry standard *PEM* file to specify the RSA Private Key that is required by the RSA 2048 signing process.

Note: For convenience during development, the software release includes a default Private Key `key.pem` file. During development, all assets are signed with the default key that is stored in the `config` directory. The default key **cannot** be used in a production system; it is not secure due to its inclusion in the release package. Contact your Intel representative for details.

Intel does not provide the key file for a production system. You must create the appropriate RSA 2048 key pair for the boot stage that the asset belongs to and provide the private half here. Key creation is a one-time process. See Chapter 3.0, “Secure Boot Overview” for details on the secure boot flow and the different keys required.

5.3.3.7 -c (optional)

Note: In the current release, assets on SD/USB support `.csbh` files only.

Use this option to create a security header that does **not** contain the input binary file. If `-c` is not used, the tool creates a `.signed` file by default.

5.3.3.8 -l (optional)

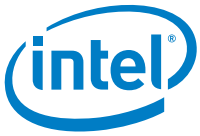
The tool reads the input file into memory to create the signature and creates the header (including the padding) in memory. To prevent unexpected behavior, the size of the output file has been limited to 1 GB.

This option disables the output file size check.

Example 5. Signing Tool Usage

```
./sign -i component1.bin -b 0x400 -s 0x01 -x 0x00 -k key.pem
```

§ §



6.0 Stage 0 (Secure Boot) Execution Details

6.1 Introduction

This chapter gives a more detailed step-by-step description of the flow of execution carried out by Stage 0 ROM code.

6.2 High Level Flow

6.2.1 System Initialization

1. System comes out of reset in 16-bit mode
2. Invalidate Cache (wbinvd)
3. Transition to Flat 32-Bit Protected mode
4. Disable cache
5. Disable NMI
6. Disable SMI
7. Read `NON-STICKY WRITE-ONCE` register.
   a. If `FORCE-CR` bit is set, issue CF9 warm reset.
8. Read `STICKY WRITE-ONCE` register.
   a. If `FORCE-WR` bit is set, issue CF9 cold reset.
9. Read `HMBOUND` register.
   a. If `Lock` bit set, write 1 to `HMBOUND_LOCK` bit of the `STICKY READ-WRITE DEBUG` register and issue CF9 warm reset.
10. Set `HMBOUND` register `HOST IO BOUNDARY` setting to equate to address 0x80080000.
11. Enable IMR Violation Reporting.
12. Set `ESRAMPGCTRL_BLOCK` register to configure all 512K eSRAM in block mode to base address 0x80000000.
13. Test the `ESRAMPGCTRL_BLOCK` register to confirm that the setting was successful.
    a. If the `BLOCK_ENABLE_PG` bit is clear, write 1 to `ESRAM_LOCK` bit of the `STICKY READ-WRITE DEBUG` register and issue CF9 warm reset.
14. Set the Stack Pointer to the top of eSRAM.

    Note: The space reserved for stack usage at this point is the top 64K of eSRAM.
15. Enable cache mode 6 for ROM address range (128K starting at 0xFFFE0000).
16. Enable cache mode 6 for sSRAM address range (512K starting at 0x80000000).
17. Write a `StackSentinel` flag into the stack space, 8K from the base of the stack space. (This sentinel is checked before transfer of execution into a Stage 1 application.)

Note: The remaining 8K of the space reserved for stack usage is reserved for Crypto Library and Debug usage.

18. Initialize Crypto Library/Debug memory.
19. Enable SPI Flash to Main Memory DMA.

6.2.2 Key Module Authentication

20. DMA Copy the *Key Module* from SPI Flash to eSRAM.
21. Validate the *Key Module* using the Authenticate Key Module function.
    a. If the *Key Module* fails to validate, go to step 36.

6.2.3 Master Flash Header Processing

22. Test if a *Master Flash Header:Identifier* is located at the relevant address in the SPI Flash.
    a. If the Identifier is not at this location, go to step 29.
23. Check *Master Flash Header:BootPriorityListCount*.
    a. If Count > Maximum Count Allowed go to step 29.

Note: The defined maximum of *Boot Priority List* items is 24, and we check that this value is valid. However, in order to speed up the boot process the Intel® Quark SoC X1000 will only process the first four items in the *Boot Priority List*. This can be seen in step 27.

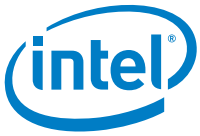
24. Check the *FlashItem* pointed to by the first entry in the *BootPriorityList*.
    a. If the Item Type is not *Host Firmware Stage 1 Signed*, go to step 27.
25. DMA Copy the *Flash Item* from the SPI Flash to eSRAM.
26. Validate the *Flash Item* using the *Authenticate Module* function.
    a. If the *Flash Item* passes validation, go to step 31.
27. Increment the count of the *Flash Items* tested.
    a. If the count >= 4 go to step 29.
28. Check the next *FlashItem* in the *BootPriorityList*.
    a. If the Item Type is *Host Firmware Stage 1 Signed*, go to step 25.
    b. If the Item Type is not *Host Firmware Stage 1 Signed*, go to step 27.

6.2.4 Fixed Location Recovery Application Validation

29. DMA Copy the *Fixed Location Recovery Application* from SPI Flash to eSRAM.
30. Validate the *Fixed Location Recovery Application* using the Authenticate Module function.
    a. If the *Fixed Location Recovery Application* passes validation, go to step 36.

6.2.5 Stage 1 Handover

31. Calculate the *Entry Point* into the authenticated module.
32. Check that the *Entry Point* falls inside the size of the Application as authenticated.
    a. If the *check* fails, go to step 36.
33. Check the *StackSentinel* has the value programmed in step 17.



a. If the *check* fails, go to step 36.

34. Jump to the *Entry Point* of the authenticated application.
35. If execution returns from the application (which it should not), continue to step 36.

6.2.6 Handling of Failure to Validate any Application

36. Store a *Fatal Error Code* in the debug memory indicating the cause for failing to validate an application.
37. Enter an infinite idle loop.

6.3 Support Functions

This section details support functions used by the Stage 0 ROM Code.

6.3.1 Authentication Functions

6.3.1.1 Authenticate Key Module

1. Test that the index of the key being authenticated is within bounds.
   a. If the test fails, return from the function with a *Fatal: Invalid Key Bank* return value.
2. Call the Authenticate Header function and check the return value.
   a. If validation fails, return from the function with a *Fatal: Key Module Validation Fail* return value.
3. Read in the *Device Key* hash value from the Intel® Quark SoC X1000 fuse bank.
4. Call the *Generate SHA 256* function to generate a hash of the *Key Modulus* field of the *Secure Boot Header*.
5. Compare the generated hash value with that read from the fuses.
   a. If the test fails, return from function with a *Fatal: Key Compare Fail* return value.
6. Call the *Generate SHA 256 Secure Module* function to generate a hash of the *Key Module*.
7. Call the *Validate RSA 2048 Signature* function passing in the generated hash and module signature.
   a. If the validation succeeds, return from the function with a *No Error* return value.
   b. If the validation fails, return from the function with a *Fatal: Key Module Validation Fail* return value.

6.3.1.2 Authenticate Module

1. Call the Authenticate Header function and check the return value.
   a. If validation fails, return from the function with a *False* return value.
2. Compare the key in the header with the key being used for validation.
   a. If the comparison fails, return from function with a *False* return value.
3. Call the *Generate SHA 256 Secure Module* function to generate a hash of the *Module*.

4. Call the *Validate RSA 2048 Signature* function passing in the generated hash and module signature.
   a. If the validation succeeds, return from the function with a *True* return value.
   b. If the validation fails, return from the function with a *False* return value.

6.3.1.3 Authenticate Header

Note: For each of the steps below, if the test fails, the function returns a *False* value. This will not be called out for each step.

1. Check that the header *Identifier* is the correct value (05F435348h).
2. Check that the header *Version* is the correct value (000000001h).
3. Check that the *Security Version Index* is within bounds (<16).
4. Check that the *Security Version Index* matches the type of module being authenticated, that is:
   a. *Key Module* requires Index 0
   b. *Stage 1 Application* requires Index 1
   c. *Fixed Address Recovery* Application requires Index 2
5. Check that the header *Security Version Number* is greater than or equal to the Intel® Quark SoC X1000 *Security Version Number* stored at the *Security Version Index*.
6. Check that the header *Hash Algorithm* is Hash 256.
7. Check that the header *Crypto Algorithm* is RSA 2048.
8. Check that the header *Signature Size* is 256 bytes.

6.3.2 Crypto Functions

The following cryptographic functions are supported in the Stage 0 code:

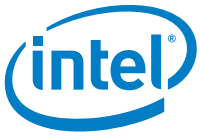
- Generate SHA 256
- Generate SHA 256 Secure Module
- Validate RSA 2048 Signature

6.4 Debug Support

The Stage 0 software, as ROM based code, cannot assume any platform related features, such as COM Port support, are present. The manner in which the ROM code indicates progress and flags errors is done via a fixed memory address being written to with progress and error codes. These codes can be read from memory using a JTAG debugger when debugging any issues with the validation of Stage 1 applications.

The fixed memory addresses used for the codes during Stage 0 code execution are:

- Progress and non-fatal errors: 8007000Ch
- Fatal Errors: 80070010h



6.4.1 Progress Codes and Non-Fatal Errors

Table 4. Progress Codes

Definition	Code Number
PROGRESS START	100
PROGRESS KEY MODULE VALID	101
PROGRESS FOUND MFH	102
PROGRESS BOOT INDEX VALID	103
PROGRESS MODULE VALID LOOP	104
PROGRESS STAGE1 SIGNED	105
PROGRESS STAGE1 NOT SIGNED	106
PROGRESS BOOT ITEM LIMIT	107
PROGRESS VALID MODULE FOUND	108
PROGRESS TRYING FIXED RECOVERY	109
PROGRESS MODULE HEADER VALIDATION PASS	120
PROGRESS RSA SIGNATURE VALID	121
PROGRESS RSA SIGNATURE INVALID	122
PROGRESS DATA SHA-256 GENERATED	123
PROGRESS SECURE MODULE SHA-256 GENERATED	124
PROGRESS START DMA COPY	125
PROGRESS DMA COPY COMPLETE	126
PROGRESS DMA COPY SIDEBAND SENT	127

Table 5. Non-Fatal Error Codes

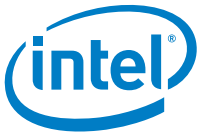
Definition	Code Number
ERROR MAGIC NUMBER FAIL	11
ERROR VERSION CHECK FAIL	12
ERROR SVN CHECK FAIL	13
ERROR HASH ALGORITHM CHECK FAIL	14
ERROR CRYPTO ALGORITHM CHECK FAIL	15
ERROR KEY SIZE CHECK FAIL	16
ERROR SIGNATURE SIZE CHECK FAIL	17
ERROR RSA KEY SIZE FAIL	18
ERROR RSA MODULUS SIZE FAIL	19
ERROR RSA EXPONENT SIZE FAIL	20
ERROR RSA MODULE VALIDATION FAIL	21
ERROR RSA KEY MISMATCH	22
ERROR INSUFFICIENT CRYPTO MEM	23
ERROR REQUIRED SVN MISMATCH	24
ERROR UNEXPECTED VAL PHASE	25
ERROR SVN INDEX OUT OF BOUNDS	26

6.4.2 Fatal Error Codes

Table 6. Fatal Error Codes

Definition	Code Number
FATAL NO VALID MODULES	1
FATAL PUNIT DMA TIMEOUT	2
FATAL MEM TEST FAIL	3
FATAL ACPI TIMER ERROR	4
FATAL UNEX RETURN FROM RUNTIME	5
FATAL ERROR DMA TIMEOUT	6
FATAL OUT OF BOUNDS MODULE ENTRY	7
FATAL MODULE SIZE EXCEEDS MEMORY	8
FATAL KEY MODULE FUSE COMPARE FAIL	9
FATAL KEY MODULE VALIDATION FAIL	10
FATAL STACK CORRUPT	11
FATAL INVALID KEYBANK NUM	12

§ §



7.0 EDKII Security

7.1 Introduction

This chapter is dedicated to describing the enhancements made to the Intel® Quark SoC X1000 UEFI firmware when built with the secure lockdown build option (see the Intel® Quark SoC X1000 Board Support Package (BSP) Build and Software User Guide in Table 1) to make it more robust and resistant to attacks and failures. These enhancements are designed to reduce the risk of a system becoming non-bootable or compromised.

7.2 Secure Boot (Secure SKU only)

The Intel® Quark SoC X1000 UEFI firmware is split into two firmware volumes (Stage1 Firmware Volume and Stage2 Firmware Volume). The Stage1 Firmware Volume continues the chain of trust by first authenticating the Stage2 Firmware Volume (via a call to the ROM code to perform the authentication) before passing control to it. This chain of trust is continued all the way up to the OS.

7.3 Isolated Memory Regions (IMRs)

The UEFI firmware uses IMRs to protect sensitive assets during and after the boot process. Please refer to the “Security Enhancements” chapter of the Intel® Quark SoC X1000 UEFI Firmware Writer’s Guide for a detailed description of the assets protected by the UEFI firmware.

Refer to Section 4.5 for the system level design using IMRs.

7.4 Legacy SPI Flash Protection

The UEFI firmware (capsule mechanism) is the only software mechanism allowed to update/recover the legacy SPI flash. In addition, the UEFI firmware is the only software module allowed to update the legacy SPI flash NVRAM area where UEFI EFI_VARIABLE_NON_VOLATILE variables are stored. To enforce this, the UEFI firmware implements SPI flash protection as described in the following sub-sections.

7.4.1 Legacy SPI Flash Range Protection

The UEFI firmware is responsible for protecting legacy SPI flash from malicious updates as early as possible in the boot flow. After protecting the legacy SPI flash ranges, the settings are locked and a reset is required to unlock. This is equivalent to the “protected boot block” mechanism supported by SPI flash chips but implemented by the legacy SPI controller.

7.4.2 Legacy SPI Flash Update Protection

The UEFI firmware is the only software module allowed to update the legacy SPI flash NVRAM area where UEFI EFI_VARIABLE_NON_VOLATILE variables are stored. However, the legacy SPI flash NVRAM area cannot be protected as in Section 7.4.1 because this area may be updated during UEFI firmware boot and also at runtime (UEFI runtime variable support). To enforce the policy of the UEFI firmware being the only software module allowed to update this area, the legacy SPI flash controller is configured to generate an SMI on any attempt to write to the legacy SPI flash. Thus UEFI firmware traps any attempts to update the legacy SPI flash.

Please refer to the “Security Enhancements” chapter of the Intel® Quark SoC X1000 UEFI Firmware Writer’s Guide for a detailed description of legacy SPI flash protection by the UEFI firmware.

7.5 PCIe Option ROMs

For added security, the UEFI firmware does not load PCIe Option ROMs from any plug in PCIe cards. Instead, any option ROMs required are built into the UEFI firmware image. This is typical for embedded systems where the onboard firmware is capable of initializing any hardware it needs.

7.6 Register Locking

Certain Intel® Quark SoC X1000 registers are responsible for setting up critical system operating features. Once set up, these registers can be locked to prevent malicious changing of their settings to compromise or hang the system. A reset is required to unlock these registers.

Please refer to the “Security Enhancements” chapter of the Intel® Quark SoC X1000 UEFI Firmware Writer’s Guide for a full list of registers locked by the UEFI firmware.

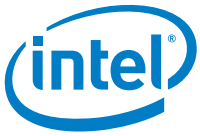
7.7 Redundant Images

It is important that corrupt legacy SPI flash images or legacy SPI flash images that fail to authenticate (secure SKU) should not leave the system un-recoverable. The UEFI firmware achieves this by providing redundant Stage1 Firmware images. Please refer to Section 3.5, “Storage of Stage 1 Public Key and Stage 1 Applications” on page 15 for a detailed description of how redundant stage1 images are implemented to further enhance the system.

Note: To find the specific placement of EDK and Bootloader Applications in Legacy SPI Flash for any particular release of software, refer to the `layout.conf` file contained within that release. See “The Layout Configuration File” on page 27 for further details.

7.8 Limiting Boot Options

The UEFI firmware only supports booting an image (OS bootloader/UEFI application) from a known location in legacy SPI flash. This image is referred to as the EDKII “Payload” and it is placed at a location in SPI flash that is known to EDKII. This removes the risk of unknown UEFI applications/drivers being loaded from USB/SD/UEFI Shell and causing unexpected system behavior or even compromising the system. For embedded systems, it is expected that the system always boots from a known image at a known location (unlike personal computers and servers).



Note that the UEFI firmware source code is capable of booting images from other media (USB/SD/UEFI Shell), but these are removed from the boot options list for this implementation to enhance security.

7.9 Denial of Service/Compromise Prevention

Some Intel® Quark SoC X1000 features have been identified as security concerns that could result in a denial of service or system compromise either accidently or maliciously. Where possible, UEFI firmware enhancements have been identified to reduce or remove these threats. For example, Intel® Quark SoC X1000 provides the ability to block the SMI pin, thus preventing SMIs for various configured events. The UEFI firmware is responsible for making sure this SMI pin remains unblocked.

Please refer to the “Security Enhancements” chapter of the Intel® Quark SoC X1000 UEFI Firmware Writer’s Guide for details of the changes made by UEFI firmware to avoid these security threats.

7.10 Memory Training Engine Lockdown

As part of DDR3 memory initialization code, the UEFI firmware makes use of a hardware engine to assist in the training of memory. Once DDR3 memory initialization is complete, UEFI firmware locks the hardware training engine to prevent further training sequences being initiated.

7.11 SMM Security Enhancements

7.11.1 SMRAM Caching

SMRAM caching is always disabled by hardware for the secure SKU regardless of the settings for the MTRRs/SMRR. This is to enhance SMM security due to caching issues. The UEFI firmware sets up SMRAM as un-cached for the non-secure SKU also to enhance its security.

As periodic SMIs are not used, SMIs in general should be very infrequent. Therefore, un-cached SMRAM is not expected to have a major system performance impact.

§ §

8.0 Bootloader Security

8.1 Introduction

The "2nd stage bootloader" reference solution carries out two important functions in terms of secure boot:

- Asset verification
  - Kernel
  - Bootloader config file - `grub.conf`
  - InitRD
- IMR setup/teardown
  - IMR setup for kernel boot parameters
  - IMR setup for compressed kernel image

This reference solution maintains a chain of trust through bootloader into the kernel by ensuring that all assets executed have been authenticated and encapsulated within an IMR.

8.2 Asset Verification

The `grub` utility verifies any kernel, `init-ramdisk` or `grub` configuration file that it relies upon in secure boot mode.

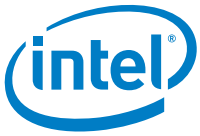
For each one of the assets kernel, `initrd` or `grub.conf` in secure boot mode, `grub` requires a corresponding signature for the associated asset.

Before trusting the contents of `grub.conf`, or executing a kernel and `initrd`, `grub` makes a call to the `grub_cln_verify_asset()` function, which validates, via Intel® Quark SoC X1000 secure boot callback, the authenticity of the relevant asset.

8.3 Isolated Memory Regions (IMRs)

Intel® Quark SoC X1000 hardware provides the capability to configure IMRs to allow/deny access by certain system agents to programmed memory ranges. Thus, an area of memory that is only for use by the host processor can be protected from other DMA agents in the system. The SoC's reference bootloader uses two IMRs so as to ensure an unbroken chain of trust from the reset vector, through bootloader and onto kernel.

Since all of the kernel code does not sit in contiguous memory, care has been taken in our reference solution, to place an IMR around both the kernel `bzImage` and the "legacy" setup code that sits in low memory.



For both IMRs used by `grub`, the flow is to set up the IMR around the relevant section of memory and subsequently load critical data into the IMR protected memory, thus ensuring no window exists when critical data is not IMR protected.

8.4 Kernel Setup and Boot Params IMR

The `grub` utility, the reference bootloader solution, sets up an IMR around the “real mode” kernel header, which in the reference solution is allocated from the address greater than 0x10000. The address that `grub` loads the compressed kernel to is determined by a call to EFI’s `AllocatePages()` BootTimeService function.

IMR1 is used for the purposes of kernel setup and boot params.

This IMR spans a contiguous range from the base address derived from the EFI provided base address to the maximum extent of the setup structure.

8.5 Compressed Kernel Image IMR

The `grub` utility, the reference bootloader solution, sets up an IMR around the compressed kernel image (`bzImage`). IMR7 is used for this purpose. The address that `grub` loads the compressed kernel to is determined by a call to EFI’s `AllocatePages()` function.

The `grub` utility sets up the IMR and then loads the data into memory.

8.6 Handoff

Once IMRs have been placed around the assets specified in the `grub.conf` file and the authenticity of the kernel and optionally `initrd` has been verified, `grub` hands control to the IMR protected `bzImage`.

From this point on, it is the responsibility of the Linux* kernel to maintain system security.

§ §

9.0 OS Security

9.1 Introduction

The reference OS solution for Intel® Quark SoC X1000 adds IMR protection to the uncompressed kernel as well as bringing the system to a final state in terms of IMR protection.

Specifically, the reference OS solution:

- Places an IMR around executable sections of the kernel image.
- Tears down any IMRs that are not required for the run-time system.
- Locks any unlocked IMRs.
- Provides a convenient debug interface to view the size, extent and state of each IMR.

9.2 Asset Verification

No specific support has been included to call back into the asset verification routine from Linux*, although there is no technical blockage to such an interface being made available.

Linux* assumes that all the assets it executes, either directly inside the kernel or via an `initrd`, have been pre-verified by the previous boot stage. Therefore, Linux* does not make any specific callback to the SoC's secure boot verification mechanism.

9.3 Early Boot IMR Support

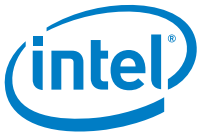
Early in the kernel boot process, before decompression takes place, an IMR is placed around the base physical address of the kernel image to the maximum memory address range, that is, 4 gigabytes.

This is necessary to ensure that the decompressed kernel runs inside of an IMR protected region.

Later phases of the kernel boot set up a smaller IMR around the run-time kernel when the necessary data to derive the correct address range becomes available.

After setting up the initial run-time kernel IMR, the early kernel boot code removes the following IMRs:

- `grub` - IMR0
- boot params - IMR1
- `bzImage` - IMR7



9.4 Run Time IMR Support

Early in the uncompressed/run-time kernel's boot path, the main IMR driver code sets up a new IMR to cover the address range of the run-time kernel, using values that only become available at this later phase in the boot process.

IMR0 is used to cover the physical address range of the kernel from the symbol "_text" to the symbol "__init_end", which represents a contiguous physical address range of executable kernel data.

At this point, the run-time IMR configuration should be:

- IMR0 - runtime kernel
- IMR1 - unused
- IRM2 - unused
- IRM3 - unused
- IRM4 - unused
- IRM5 - Legacy S3 memory
- IMR6 - ACPI NVS, Run Time Code, Run Time Data, Reserved memory, ACPI reclaim memory
- IMR7 - unused

9.5 Debug Interface

For the purposes of system debug, an interface is provided in `/proc` to view the setup of the IMRs on a booted reference Intel® Quark SoC X1000 system.

Read data from `/proc/driver/imr` to view the address range of each IMR[0-7] and its state, in the run time system.

§ §

Appendix A Master Flash Header (MFH) Data Structure

A.1 MFH Format

Table 7. Master Flash Header Format

Offset (hex)	Byte 3	Byte 2	Byte 1	Byte 0
0x00	Intel® Quark SoC X1000 MFH Identifier			
0x04	Version			
0x08	Flags			
0x0C	Next Header Block			
0x10	Flash Item Count m			
0x14	Boot Priority List Count n			
0x18	Boot Index 0			
0x1C	Boot Index 1			
	...			
0x18 + 0x04(n-1)	Boot Index n-1			
0x18 + 0x04(n)	Flash Item 0 [0x10 Bytes]			
0x18 + 0x04(n) + 0x10	Flash Item 1 [0x10 Bytes]			
	...			
0x18 + 0x04(n) + 0x10(m-1)	Flash Item m-1 [0x10 Bytes]			

Note: The MFH is in little-Endian format.

A.1.1 MFH Identifier

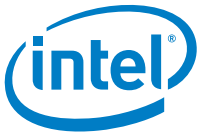
The Identifier is a fixed value for identifying the Master Flash Header. Its value is always set to 05F4D4648h corresponding to "_MFH" in ascii characters. This can be used to speed up parsing by means of quickly detecting if a MFH is not present.

A.1.2 Version

The Version field indicates the revision of the Master Flash Header is present. If the format of the Master Flash Header changes, this number is incremented. The current version is 00000001h.

A.1.3 Flags

32 bits are reserved in the Master Flash Header for later use as flags. All bits are currently reserved and should be written as 0.



A.1.4 Next Header Block

Reserved for future use, where multiple MFHs can be used should the first MFH have insufficient space to describe all items in the Flash device. This should be set to 00000000h.

A.1.5 Flash Item Count (m)

The Flash Item Count field indicates how many flash items are contained within this header block. This is the value **m** in the Master Flash Header Format table.

A.1.6 Boot Priority List Count (n)

The Boot Priority List Count field indicates how many bootable flash items are present. This is the value **n** in the Master Flash Header Format table, and it is always equal or less than the number of Flash Items m.

A.1.7 Boot Index (0..n)

The Boot Index fields are the n 32-bit values following the Boot Priority List Count field, where n is the value contained in Boot Priority List Count. Each value is the index, starting from 0, of a flash item in this header block that describes a host firmware application from which to boot.

A.1.8 Flash Item (0..m)

The Flash Item fields describe the location and attributes of various items that are stored in flash. Following is a description of the Flash Item; each instance in the Master Flash Header follows this format.

Table 8. Flash Item Format

Offset (hex)	Byte 3	Byte 2	Byte 1	Byte 0
0x00	Type			
0x04	Flash Item Address			
0x08	Flash Item Length			
0x0C	Rsvd			

A.1.8.1 Type

The Type field indicates the contents of the elements described by this Flash Item. The value will be one of the values contained in the below table.

Table 9. List of Types (Sheet 1 of 2)

Value	Type	layout.conf Type	Notes
0x00000000	Host Firmware Stage 1	host_fw_stage1	Eg. EDKII PEI phase
0x00000001	Host Firmware Stage 1 Signed	host_fw_stage1_signed	A signed stage 1
0x00000002	rsvd		
0x00000003	Host Firmware Stage 2	host_fw_stage2	e.g. EDKII DXE phase

Table 9. List of Types (Sheet 2 of 2)

Value	Type	layout.conf Type	Notes
0x00000004	Host Firmware Stage 2 Signed	host_fw_stage2_signed	A signed stage 2
0x00000005	Host Firmware Stage 2 Configuration	mfh.host_fw_stage2_conf	
0x00000006	Host Firmware Stage 2 Signed Configuration	mfh.host_fw_stage2_conf_signed	Signed stage 2 configuration
0x00000007	Host Firmware Parameters	mfh.host_fw_parameters	BIOS settings, “CMOS”, etc. read/write settings
0x00000008	Host Recovery Firmware	mfh.host_recovery_fw	Recovery application used in case of asset corruption.
0x00000009	Host Recovery Firmware Signed	mfh.host_recovery_fw_signed	
0x0000000A	rsvd		
0x0000000B	Bootloader	mfh.bootloader	Grub etc.
0x0000000C	Bootloader Signed	mfh.bootloader_signed	
0x0000000D	Bootloader Configuration	mfh.bootloader_conf	Grub.conf for example
0x0000000E	Bootloader signed configuration	mfh.bootloader_conf_signed	
0x0000000F	rsvd		
0x00000010	Kernel	mfh.kernel	Linux* kernel, etc.
0x00000011	Kernel Signed	mfh.kernel_signed	A signed Kernel
0x00000012	RAM Disk	mfh.ramdisk	
0x00000013	RAM Disk Signed	mfh.ramdisk_signed	
0x00000014	rsvd		
0x00000015	Loadable Program	mfh.loadable_program	An executable run by host firmware (e.g. EDK payload)
0x00000016	Loadable Program Signed	mfh.loadable_program_signed	An executable run by host firmware (e.g. EDK payload)
0x00000017	rsvd		
0x00000018	Build Information	mfh.build_information	Free form, human-readable description of the build contents.

A.1.8.2 Flash Item Address

This is the absolute address of the flash item in the system’s overall 4G address space. In the Intel® Quark SoC X1000 Secure SKU, the SPI flash contents shall be located just below the 4G boundary.

A.1.8.3 Flash Item Length

This field indicates the length in bytes of the given flash item.

A.1.8.4 Reserved Field

This is a reserved field for future use. An example is storing the version number of a flash item.

§ §

Appendix B Secure Boot Header Data Structures

B.1 Overview

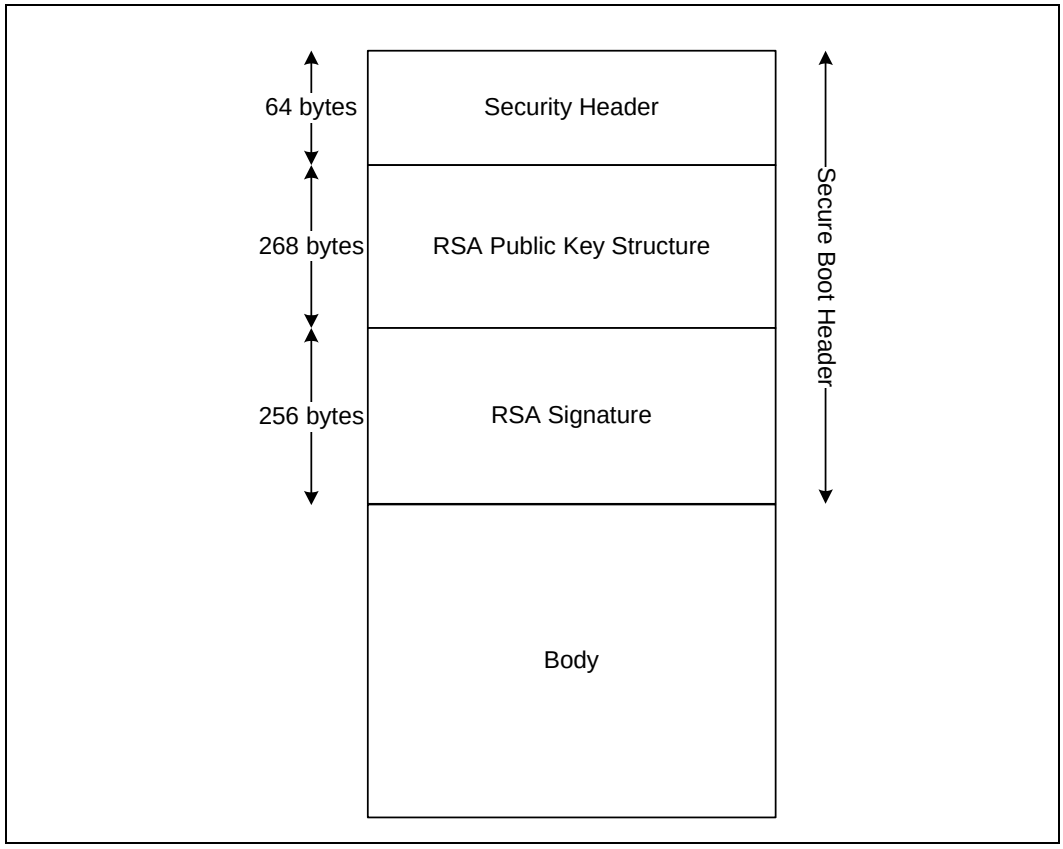
A Secure Boot Header (SBH) is prepended to an asset during the signing process to create a *Signed Module*. Within a signed module, the asset can also be referred to as the *body* of the module.

The Secure Boot Header is made up of three individual components:

- Security Header
- RSA Public Key
- RSA Signature

Figure 5 shows a signed module and includes the three components of an SBH.

Figure 5. Signed Module Layout



B.2 Security Header Data Structure

Table 10. Security Header Data Structure

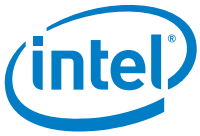
Byte Offset	Name	Description
0x0	Identifier (4 bytes)	Value must be set to 0x5F435348 corresponding to "_CSH" in ASCII.
0x4	Version (4 bytes)	Version of the Security Header definition used, must be 00000001h for Intel® Quark SoC X1000.
0x8	Module Size in bytes (4 bytes)	Size of the entire module. Module Size = (Size of SH (128 bytes) + Size of Signature (256 bytes) + Public Signature Key (260 bytes)+ Size of Body (multiple of 64 bytes).
0xC	Security Version Number Index (4 bytes)	Index of the SVN to use for validation of signed module.
0x10	Security Version Number (4 bytes)	Value used to prevent Roll-back prevention of software modules.
0x14	Reserved Module ID (4 bytes)	Currently unused for Intel® Quark SoC X1000.
0x18	Reserved Module Vendor (4 bytes)	Vendor identifier. Set to 00008086 for Intel® Quark SoC X1000.
0x1C	Reserved Date (4 bytes)	BCD representation of signing date as yyyymmdd, where yyyy=4 digit year, mm=1-12, dd=1-31. Currently unused.
0x20	Module Header Size in Bytes (4 bytes)	Total length of the header including the crypto fields for the public key and signature in bytes.
0x24	Hashing Algorithm (4 bytes)	Hashing algorithm used for signing. SHA-256 = 00000001h
0x28	Crypto Algorithm (4 bytes)	Crypto algorithm used for signing. RSA 2048 = 00000001h
0x2C	Key Size in Bytes (4 bytes)	Logical Size of key in bytes.
0x30	Signature Size Bytes (4 bytes)	Total length of the signature including any padding optionally added.
0x34	Reserved Next Header pointer (4 bytes)	32-bit pointer to the next secure boot module in chain of trust. Currently unused.
0x38 - 0x3F	Reserved	Reserved for future use; must be all zeros.

B.3 RSA Public Key Data Structure

The RSA Key structure immediately follows the Security Header; it contains key description fields and the key itself, and conforms to the following specification for Intel® Quark SoC X1000 A0.

Table 11. RSA Key Structure

Byte Offset	Name	Description
0x0	Modulus Size Bytes (4 bytes)	Size of the Public Key Modulus in bytes.
0x4	Exponent Size Bytes (4 bytes)	Size of the public key exponent in bytes.
0x8	Modulus (256 bytes)	Public key modulus data.
0x108	Exponent (4 bytes)	Public key exponent data.



B.4 Security Version Number Indexing

The Security Header (SH) contains a Security Version Number (SVN) Index, as documented above. This index allows the authentication function to determine which SVN to use as part of authentication.

Each asset is free to use whatever index it wishes with the following restrictions enforced by Stage 0 ROM code: indices 0, 1, and 2 are reserved for the *Key Module*, *Stage 1 Applications,* and *Fixed Location Recovery Application* respectively and must be used for those application types.

Reference SVN index values, as used in Intel software are listed below.

The maximum SVN index supported is 15 (16 SVNs in total).

Table 12. SVN Index Allocation

SVN Index	Fixed by Hardware	Asset
0	Y	Key Module
1	Y	EDKII Stage 1 firmware application
2	Y	Fixed Location EDKII Recovery application
3	N	EDKII Stage 2 firmware application
4	N	Bootloader
5	N	Bootloader Config
6	N	Kernel
7	N	Ramdisk
15	N	Update capsule ***Note:*** The update capsule itself is checked against SVN index 15. Capsule contents have their own SVN which is also checked. For example, a Bootloader payload will be checked against SVN index 4.

§ §

Appendix C Firmware Volume Overview

C.1 Definition and Layout

A Firmware Volume (FV), as defined by the EFI PI specification (http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK_II_Specifications), is a logical firmware device. A FV requires a file system, the default one being the Firmware File System (FFS).

An FV, along with one or more file systems, enables a file level interface to FLASH storage.

The general FV Layout is shown in the following table, further details can be found on the site referenced above.

Table 13. General FV Layout (Sheet 1 of 2)

0	15	16	31
Zero Vector			
Zero Vector			
Zero Vector			
Zero Vector			
File System GUID			
File System GUID			
File System GUID			
File System GUID			
Size			
Size			
Signature (“_FVH”)			
Attributes			
Header Length		Checksum	
Extended Header Offset		Reserved	Revision
First Block Map Entry Num Blocks			
First Block Map Entry Length			
...			
N Block Map Entry Num Blocks			
N Block Map Entry Length			
FV Name GUID			
KEY:	FV Header	Optional FV Extended Header	FFS Files

Table 13. General FV Layout (Sheet 2 of 2)

0	15	16	31
FV Name GUID			
FV Name GUID			
FV Name GUID			
Extended Header Size			
First Extended Entry Size			
First Extended Entry Type			
...			
N Extended Entry Size			
First Extended Entry Type			
N number of FFS Files			
KEY:	FV Header	Optional FV Extended Header	FFS Files

C.2 Tools

EDK II provides a Toolchain for processing EDK II content as a separate project called “BaseTools”, among which are a set of FV generation tools. BaseTools are available under BSD license and can be run on Windows*/MacOS*/Linux*.

Tools required for FV generation/parsing are:

- **GenFw** – Genfw is mainly used to process a PE32 image to get the expected image data or image file. PE32 is a general-purpose image format that contains, among other information, data identifying the target environment for execution of the image.
- **GenSec** – GenSec generates valid EFI_SECTION type files that conform to the firmware file section defined in the PI specification, from PE32/PE32+/COFF image files or other binary files. This utility produces a file that is the section header concatenated with the contents of the input file. It does not validate that the contents of the input file match the section added.
- **GenFfs** – GenFfs generates FFS files for inclusion in a firmware volume. FFS file is the file system file for the firmware storage defined in Volume 3 of the PI 1.0 specification. This utility aggregates all of the file components into a single, correctly formed FFS file.
- **GenFv** – GenFv generates a PI firmware volume image or a UEFI capsule image from the PI firmware files or the binary files that conforms to the firmware volume image format defined in PI specification or uefi capsule image format defined in the UEFI specification.
- **VolInfo** – Displays the contents of a firmware volume

C.2.1 Using FV Tools

To manually wrap a module with a firmware volume, the following tools must be used in the specified order:

- **GenFw**
- **GenSec**

- **GenFfs**
- **GenFv**

The following example demonstrates the manual FV wrap process on a `grub.efi` module:

Example 6. Sample FV Wrapping Process

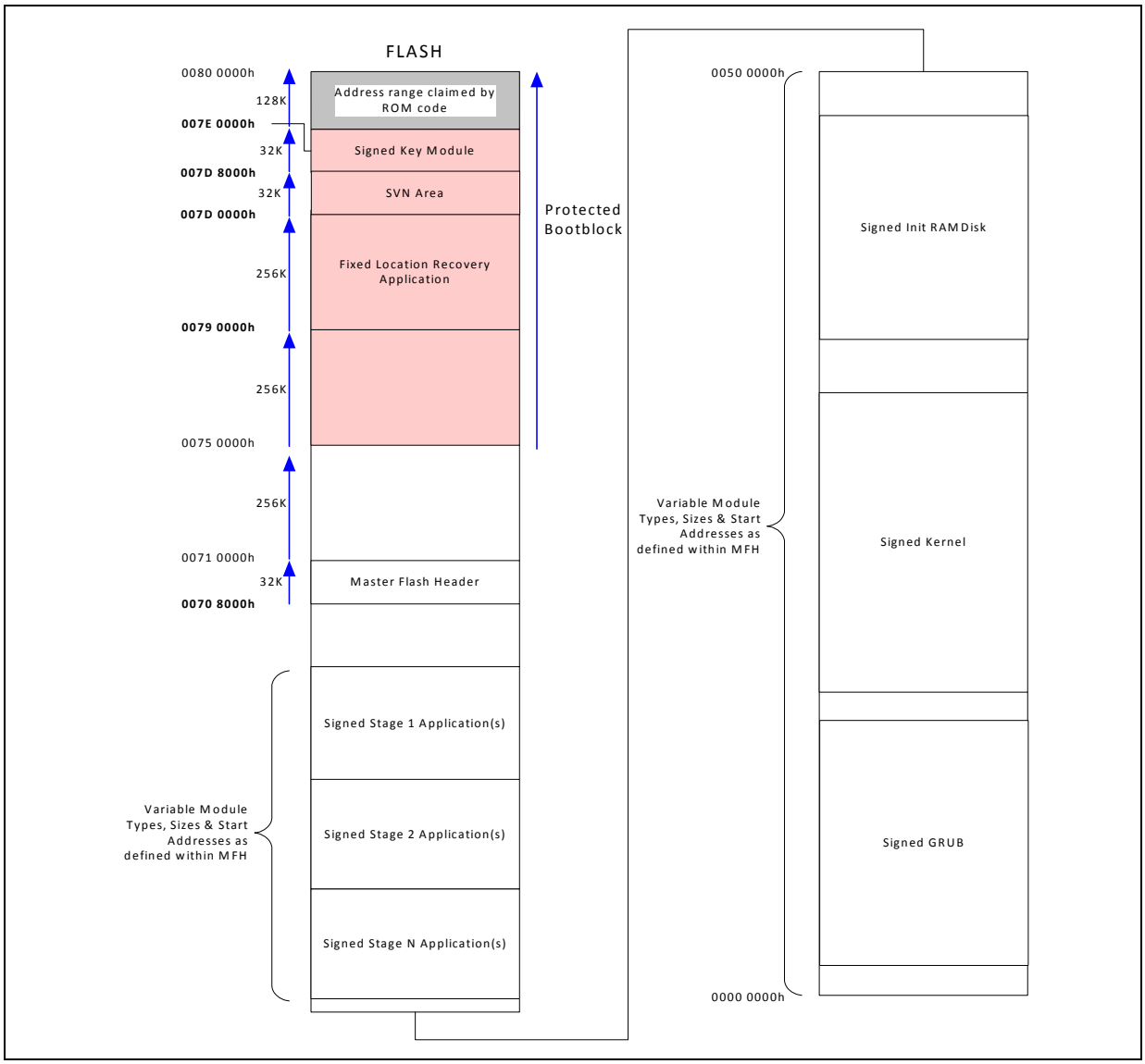
```
GenFw grub.efi -o grub.efi.fw -e UEFI_APPLICATION

GenSec -s EFI_SECTION_PE32 grub.efi.fw -o grub.efi.fw.pe32

GenFfs -o image.ffs -t EFI_FV_FILETYPE_APPLICATION -g B43BD3E1-64D1-4744-9394-
D0E1C4DE8C87 -i grub.efi.fw.pe32

GenFv  -i inf/grub.efi.inf -o grub.efi.fv
```

§ §

Appendix D Sample Flash Layouts

Figure 6 depicts a standard generic 8 MB SPI-flash layout. The addresses within the diagram are relative to the start of SPI flash device address range.

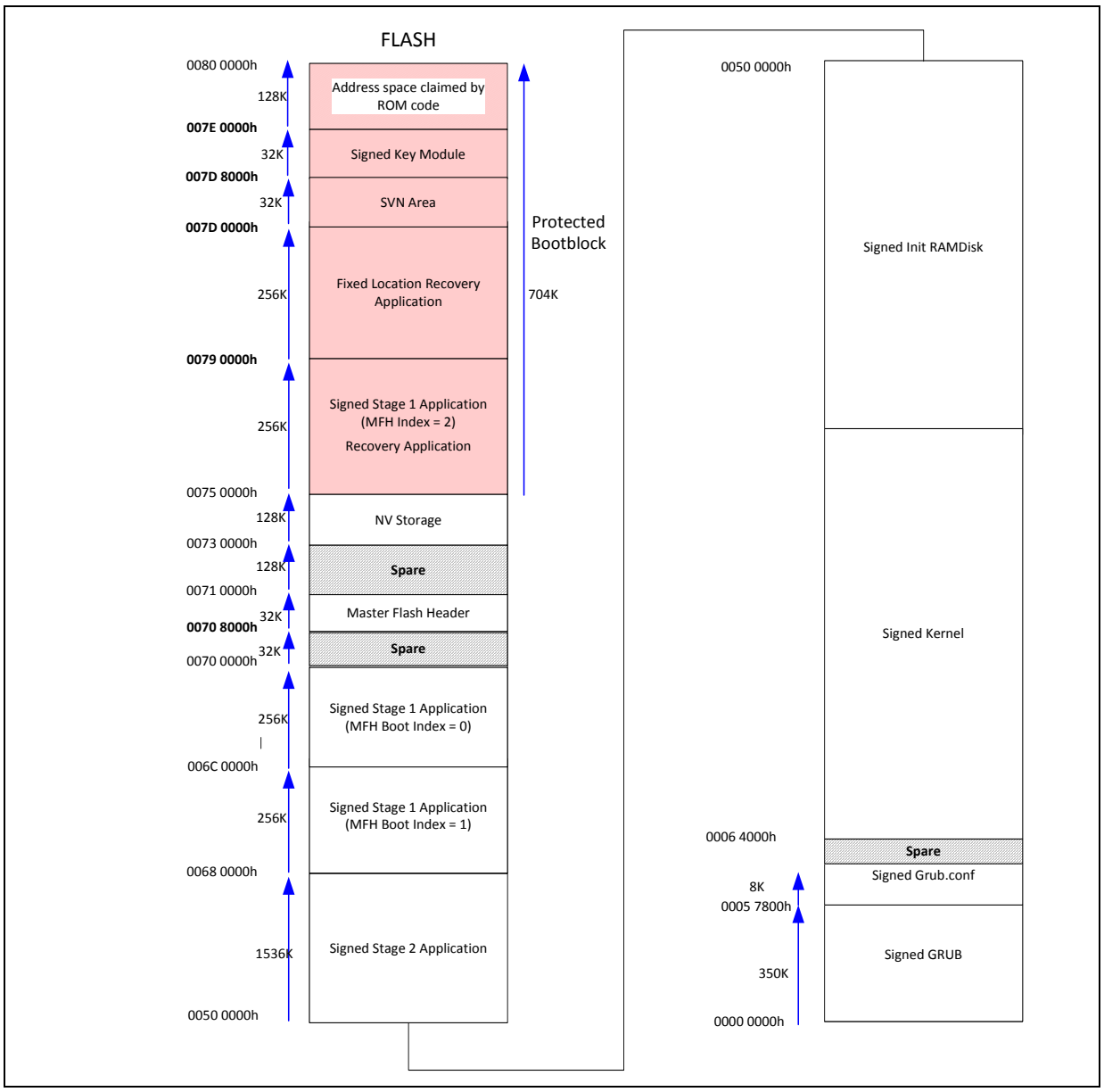
Figure 6. Generic 8 MB Flash Layout



Note: Any addresses in **bold** are fixed addresses.

Figure 7 depicts a example 8 MB SPI-flash layout.

Figure 7. Example 8 MB Flash Layout

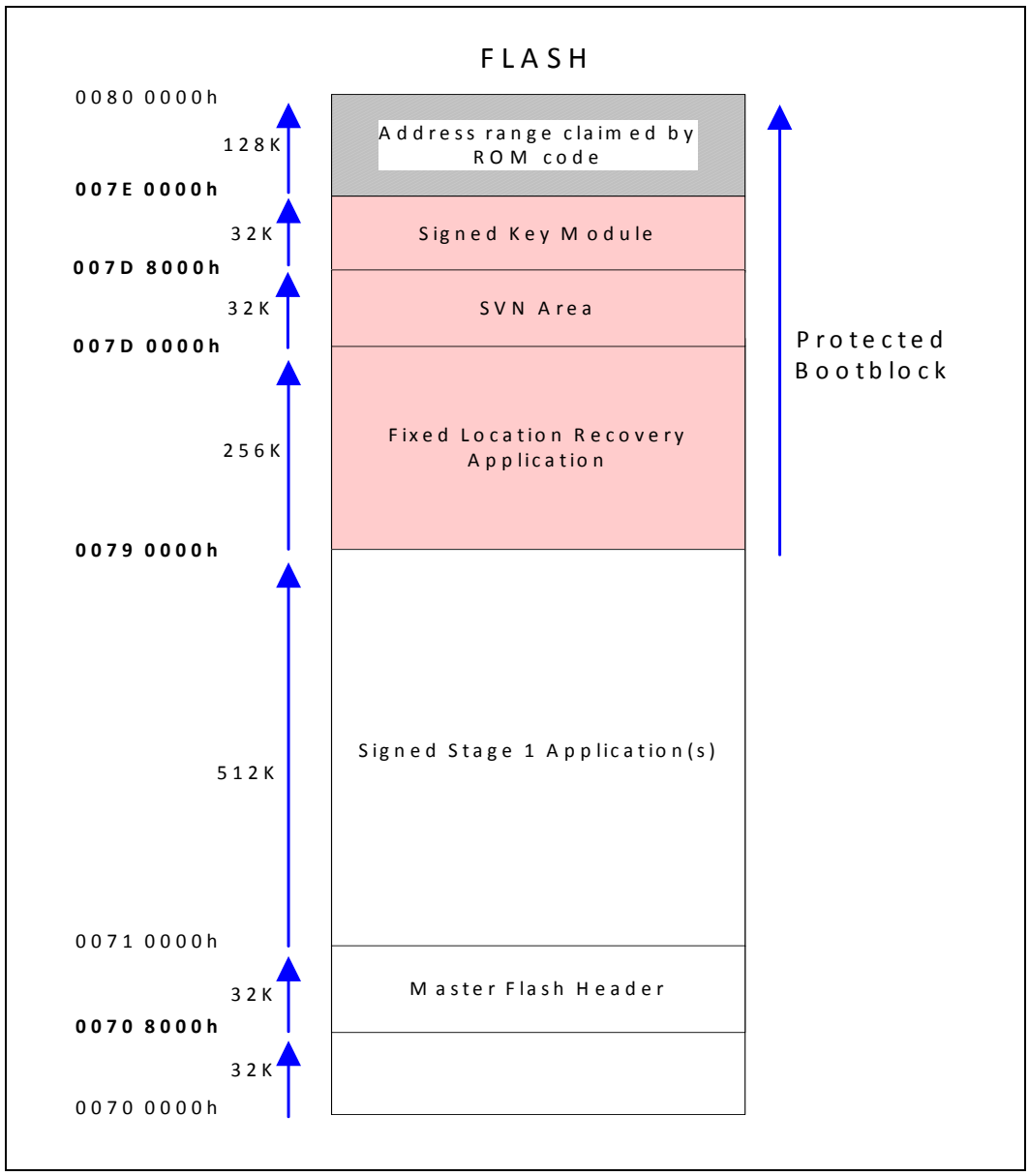


Note: Any addresses in **bold** are fixed addresses.

Note: The image sizes depicted here for Firmware, for example, Recovery, Stage1, Stage2 have reserved enough space to allow an image to grow if required. Images much smaller in size may well be used in the final product.

Figure 8 depicts a generic 1 MB flash layout.

Figure 8. Generic 1 MB Flash Layout



Note: Any addresses in **bold** are fixed addresses.

§ §