Intel® Stratix® 10 Device Security User Guide

Updated for Intel® Quartus® Prime Design Suite: **19.3**

Subscribe

Send Feedback

UG-S10SECURITY | 2020.01.15

Latest document on the web: PDF | HTML

Contents

1. Intel® Stratix® 10 Device Security Overview....................4
1.1. Intel Stratix 10 Secure Device Manager (SDM).................... 6
1.2. Enabling Intel Stratix 10 Security Features....................7
1.2.1. Side Channel Mitigation.................... 7
1.3. Owner Security Keys and Programming.................... 8
1.3.1. Owner Root Public Key Hash Programming....................9
1.3.2. AES Root Key Programming.................... 9
1.4. Planned Security Features....................9
1.4.1. Physically Unclonable Function (PUF) Overview....................9
1.4.2. Anti-Tampering....................10
1.4.3. Black Key Provisioning....................10

2. Design Authentication....................11
2.1. The Configuration Bitstream 11
2.2. Signature Block.................... 13
2.2.1. Canceling Intel Firmware ID.................... 16
2.2.2. Authentication for HPS Software.................... 17

3. Using the Authentication Feature....................18
3.1. Step 1: Creating the Root Key.................... 19
3.2. Step 2: Creating the Design Signing Key....................19
3.3. Step 3: Appending the Design Signature Key to the Signature Chain....................20
3.4. Step 4: Signing the Bitstream....................21
3.5. Step 4a: Signing the Bitstream Using the Programming File Generator....................21
3.6. Step 4b: Signing the Bitstream Using the quartus_sign Command....................23
3.7. Step 5: Programming the Owner Root Public Key for Authentication....................24
3.8. Step 5a: Programming the Owner Root Public Key....................24
3.9. Step 5b: Calculating the Owner Root Public Key Hash.................... 26

4. Co-Signing Device Firmware Overview....................27
4.1. Using the Co-Signing Feature.................... 27
4.1.1. Prerequisites for Co-Signing Device Firmware.................... 28
4.1.2. Generating the Owner Firmware Signing Key.................... 28
4.1.3. Co-Signing the Firmware 29
4.1.4. Powering On In JTAG Mode After Implementing Co-Signed Firmware....................29

5. HPS Debug Using a Certificate.................... 30
5.1. Enabling HPS JTAG Debugging 31

6. Signing Command Detailed Description.................... 33
6.1. Generate Private PEM Key....................34
6.2. Generate Public PEM Key....................34
6.3. Generate Root Signature Chain....................34
6.4. Append Key to Signature Chain....................35
6.5. Sign the Bitstream, Firmware, or Debug Certificate.................... 36
6.6. Calculate Root Public Key Hash from QKY....................36

7. Encryption and Decryption Overview.................... 37
7.1. Using the Encryption Feature....................39

7.1.1. Step 1: Preparing the Owner Image and AES Key File.................................. 39
7.1.2. Step 2a: Generating Programming Files Using the Programming File Generator.. 40
7.1.3. Step 2b: Generating Programming Files Using the Command Line Interface 41
7.1.4. Step 3a: Specifying Keys and Configuring the Encrypted Image Using the Intel Quartus Prime Programmer 41
7.1.5. Step 3b: Programming the AES Key and Configuring the Encrypted Image Using the Command Line.. 44
7.1.6. Storing the AES Key AES in Physical eFuses...45
7.1.7. Storing the AES Key in BBRAM using the JTAG Mailbox.................................45

8. Encryption Command Detailed Description.. 46

8.1. Make AES Key.. 46
8.2. Encrypt the Bitstream..47

9. Using eFuses 48

9.1. Fuse Programming Input Files.. 50
9.1.1. Fuse File Format... 51
9.1.2. Programming eFuses 51
9.1.3. Canceling eFuses.. 53
9.1.4. Converting Key, Encryption, and Fuse Files to Jam Staple File Formats........... 53

10. Document Revision History for Intel Stratix 10 Device Security User Guide...............55

1. Intel® Stratix® 10 Device Security Overview

Intel® Stratix® 10 devices provide flexible and robust security features to help protect sensitive data, intellectual property, and the device itself under both remote and physical attacks.

Intel Stratix 10 devices provide two main categories of security features: authentication and encryption.

Authentication helps to ensure that both the firmware and the configuration bitstream are from a trusted source. Authentication is fundamental to Intel Stratix 10 security. You cannot enable any other Intel Stratix 10 security features without enabling owner authentication.

Encryption helps to protect confidential information such as intellectual property or sensitive data from being extracted from the owner configuration bitstream.

Here are the specific security features that Intel Stratix 10 devices provide:

Authentication Category

- Elliptic Curve Digital Signature Algorithm (ECDSA) Based Public-Key Authentication: Intel Stratix 10 devices always require firmware authentication for all Intel firmware that loads into silicon. The ECDSA authentication of firmware implements this requirement. Intel is the only source that provides the primary firmware for the Secure Device Manager (SDM) and all other firmware that runs on other configuration processors in the Intel Stratix 10 device.

  Intel Stratix 10 devices do not require authentication for configuration bitstreams. You may enable configuration bitstream authentication by programming the hash of your root public key into eFuses. This process establishes you as the owner of the device. After you enable configuration bitstream authentication, you must create a valid signature chain based on your root key for each configuration bitstream. Your Intel Stratix 10 device completes configuration after successful validation of your signature chain.
- Anti-tampering security feature: Anti-tampering addresses physical attacks on silicon. There are two categories of anti-tampering features: passive and active anti-tampering.
  - The passive anti-tampering feature enforces physical security features using redundancy and interlocking systems. Passive anti-tampering is always running on Intel Stratix 10 devices. Passive anti-tampering functions do not operate in response to a particular function.
  - Active anti-tampering responds when the silicon detects physical attacks from the outside. By default, all active anti-tampering functions are off. When the active anti-tampering function is on, you can select which detection functions and responses to enable. Active anti-tampering is planned for a future release. Refer to Anti-Tampering on page 10 for more information.

Encryption Category

- Advanced Encryption Standard (AES)-256 encryption: This feature helps protect the confidentiality of intellectual property (IP) or sensitive data in the owner configuration bitstream. AES-CTR (counter) mode is the base for bitstream encryption. To reduce AES key exposure AES decryption only operates on data that has already passed public key authentication.
- Side channel protection: This feature helps to protect the AES Key and confidential data from extraction through non-intrusive attacks. Intel Stratix 10 devices include the following functions to minimize any potential side channel leakage:
  - The authentication first flow helps to protect against encrypted bitstream modifications that reveal an encryption key.
  - A key update function reduces the amount of bitstream data encrypted with a single key.
  - Long route data line scrambling reduces the exposure of decrypted configuration data on the chip-wide configuration network.
  - A 256-bit wide direct key bus loading minimizes the transmission time of sensitive key material.
  - Key scrambling limits any potential side-channel exposure when you store the AES root key in eFuses.
- Multiple AES root key choices: Intel Stratix 10 devices currently support two different locations for root AES keys: eFuse and BBRAM. In addition, physically unclonable function (PUF) is planned for a future release. Refer to Physically Unclonable Function (PUF) Overview on page 9 for more information.

These security features are available in Intel Stratix 10 devices that support advanced security. The following table lists the security features that Intel Stratix 10 devices support.

Intel Stratix 10	Authentication	Advanced Security
GX	Yes	-AS suffix devices
SX	Yes	-AS suffix devices
MX	Yes	-AS suffix devices
TX	Yes	-AS suffix devices
DX	Yes	Yes

Related Information

- Intel Stratix 10 Device Security User Guide Archives
- Intel Quartus® Prime Pro Edition User Guide Programmer
  Describes operation of the Intel Quartus® Prime Pro Edition Programmer which allows you to configure Intel FPGA devices and program CPLD and configuration devices via connection with an Intel FPGA download cable.
- Intel Stratix 10 Device Feature Status
  For more information about the status of planned Intel Stratix 10 device security features.

1.1. Intel Stratix 10 Secure Device Manager (SDM)

The Secure Device Manager (SDM) is a triple-redundant processor-based module that manages the configuration and security features of Intel Stratix 10 devices. The SDM authenticates and decrypts configuration data.

Figure 1. Secure Device Manager

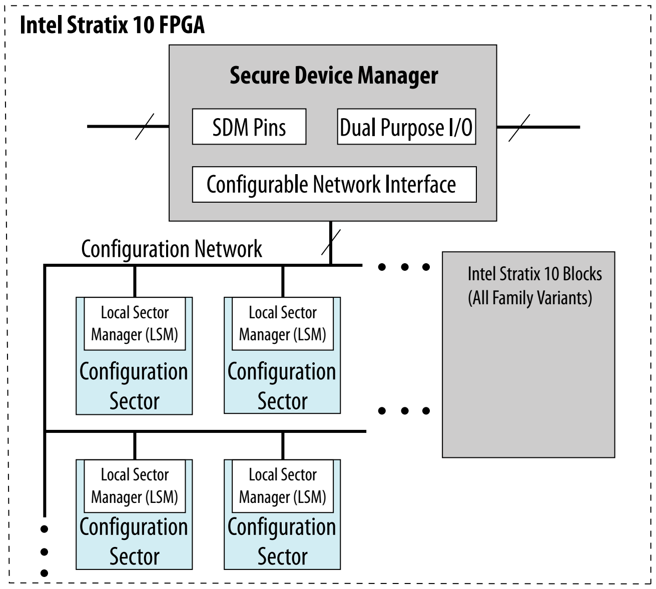

Intel Stratix 10 FPGA
Secure Device Manager
SDM Pins
Dual Purpose I/O
Configurable Network Interface
Configuration Network
Intel Stratix 10 Blocks (All Family Variants)
Local Sector Manager (LSM)
Configuration Sector
Local Sector Manager (LSM)
Configuration Sector
Local Sector Manager (LSM)
Configuration Sector
Local Sector Manager (LSM)
Configuration Sector

Figure 2. Secure Device Manager

Secure configuration includes the following steps:

- If you have enabled authentication, the SDM checks that a trusted source, the device owner, has authorized the configuration bitstream.
- The SDM always performs an integrity check over the bitstream using SHA-256 or SHA-384. This integrity check protects against intentional attacks and against accidental corruption of the bitstream, such as a bad write to flash.
- If the configuration bitstream authenticates and you have enabled AES Encryption, the SDM decrypts the data. The SDM drives the decrypted data on the configuration network to Local Sector Managers (LSM) on the configuration network. Each LSM parses the sector configuration block data and configures the logic elements in the sector that it manages.

Related Information

Intel Stratix 10 Configuration User Guide: Secure Device Manager

Send Feedback

1.2. Enabling Intel Stratix 10 Security Features

Enabling any of the Intel Stratix 10 device security features first requires you to program the owner root public key hash into eFuse storage. Programming the hash of the root public key enables authentication, after which your configuration bitstreams must be signed. In addition, other security features, such as bitstream encryption, are available. Intel Stratix 10 devices support both virtual and physical eFuse programming. Before you program any security eFuse, Intel recommends that you use the virtual eFuse programming to test that the values being programmed are correct.

Caution: Incorrect programming of security eFuses can permanently prevent the device from configuring.

The fusing process automatically computes the hash of the owner root public key. When you program the owner root key hash, the programmer automatically programs the hash value, not the full key.

You can enable the following additional security options to further enhance the security level:

- Advanced Encryption Standard (AES) Encryption protects your IP and secures your data. This option includes multiple sub-options relating to side channel mitigation.
- Configuration firmware joint signature capability specifies that you, in addition to Intel, must sign the version of configuration firmware that runs on your device. If you enable the joint signature capability, the device only loads firmware signed by both Intel and by you, the device owner. An eFuse on the Intel Stratix 10 device enables this feature. For a full list of available eFuse security options, refer to *Using eFuses*.

eFuse programming sets a minimum-security strength. All eFuse enforced security options are permanent.

In contrast to permanent security features, Intel Stratix 10 devices include some dynamic security options that you can control without using eFuses. Disabling HPS debugging is one example of a dynamic security feature. You control dynamic security options by setting optional fields in the configuration bitstream. The Intel Stratix 10 device enforces dynamic security options beginning with bitstream configuration, instead of at power-on, providing additional flexibility.

Related Information

Using eFuses on page 48

1.2.1. Side Channel Mitigation

Side channel mitigation technology helps prevent secret leakage from the Intel Stratix 10 device. Side channel mitigation is not limited to the AES engine. Any circuit which could transport secret key material has its associated mitigation.

The following side channel mitigation features are available in Intel Stratix 10 devices:

- Authentication first: The device authenticates the bitstream before decrypting it. Attackers cannot perform differential attacks on the AES encrypted data without breaking authentication.
- Key update: Limits the amount of encrypted data per key to 1024 bytes.
- Direct key loading: Uses a 256-bit point-to-point key bus to reduce emissions.
- Data scrambling: Scrambles data on long wires within the configuration network on a chip (NoC).

1.3. Owner Security Keys and Programming

Intel Stratix 10 devices support two types of security keys:

- Owner root public key hash: Programming this key enables the owner configuration bitstream authentication. Configuration bitstream authentication is the fundamental security feature. You must enable configuration bitstream authentication before you can enable other security features. The Intel Stratix 10 device stores the SHA-256 or SHA-384 hash of this key in physical eFuses or virtual eFuses. This hash validates the integrity of the root public key, which is the first step in the process to authenticate the configuration bitstream.
- Owner AES key: This optional key decrypts the encrypted owner image during the configuration process. You can store the AES key in virtual eFuses, physical eFuses, or a BBRAM. PUF support for AES key handling is planned for a future release.

  In contrast to eFuse (non-volatile) storage, BBRAM storage is reprogrammable. The BBRAM key vault holds a single key. Programming a new key deletes the previously programmed key. The BBRAM key vault includes a built-in function to perform periodic key flipping to prevent key imprinting. The BBRAM has its own power supply. V_{CCBAT} powers the BBRAM AES key. The voltage range is 1.2V - 1.8V. For more information about required voltage ranges refer to the *Intel Stratix 10 Device Family Pin Connection Guidelines*.

  You program both the root public key hash and the AES key using JTAG. The configuration bitstream specifies the owner AES key location. For extra security, you can program fuses to disable some of the key storage locations. For example, if your design stores the AES key in eFuses, you can program the BBRAM root key disable fuse for additional security.

  Intel Stratix 10 devices support both red key (unencrypted) and black key (encrypted) provisioning (transport). JTAG transmits keys in an unencrypted format. Encrypting the AES key reduces the risk of disclosing the key during the manufacturing process. Refer to Black Key Provisioning on page 10 for more information about programming an encrypted AES key.

Note: You program or blow eFuses by flowing a large current for a specific amount of time. This process is irreversible.

Related Information

- Recommended Operating Conditions for V_{CCBAT} in Stratix 10 Device Datasheet
- Intel Stratix 10 Device Family Pin Connection Guidelines

1.3.1. Owner Root Public Key Hash Programming

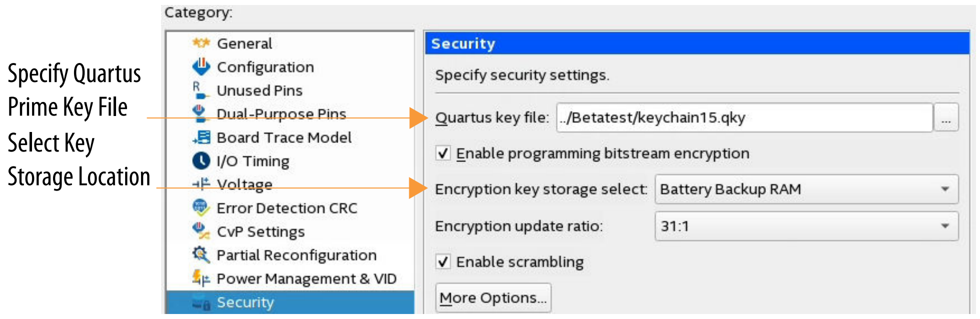
You can store the owner root public key hash in virtual eFuses (volatile) or physical eFuses (non-volatile).

You specify either virtual or physical eFuses when you program your device. Once you program the physical eFuse key, you cannot change or reprogram the key.

1.3.2. AES Root Key Programming

You specify the storage option for the AES root key on the **Security** page of the **Assignments ➤ Device ➤ Device and Pin Options**. In the current release, you can select **Battery Backup RAM (BBRAM)** or **eFuses**. When you generate the SRAM Object File `.sof` the Intel Quartus® Prime Pro Edition Software records the key you specify to partially encrypt the configuration bitstream.

Figure 3. Specify Storage Location for Encryption Key



Specify security settings.

Quartus key file: ../Betatest/keychain15.qky

✓ Enable programming bitstream encryption

Encryption key storage select: Battery Backup RAM

Encryption update ratio: 31:1

✓ Enable scrambling

The Intel Quartus Prime Programmer also includes an **Encryption Key Select** option with two choices: **Battery Backup RAM** or **eFuses**. This option is available for Intel Stratix 10 and later devices that include the SDM when you program a Intel Quartus Prime encryption key `.qek`.

1.4. Planned Security Features

Some Intel Stratix 10 advanced security features are not currently supported, but are planned to be supported in a future release. These features include support for a PUF, anti-tampering, and black key provisioning.

1.4.1. Physically Unclonable Function (PUF) Overview

The Intel Stratix 10 device provides access to the PUF as part of the device configuration process. The PUF generates device-unique, unclonable keys based on SRAM initialization patterns. You can use the PUF to assist with AES root key encryption. Encrypting an AES key is also called key wrapping. You store the wrapped AES root key in external flash memory. Using the PUF also requires storing PUF helper data in the external flash memory.

Note: To enable the PUF function, you must negotiate a license agreement with Intrinsic ID.

Related Information

Intel Stratix 10 Device Feature Status
For more information about the status of planned Intel Stratix 10 device security features.

1.4.2. Anti-Tampering

Anti-tampering features help detect and respond to certain physical attacks on silicon.

The SDM monitors operating conditions such as input clocks, voltage, and temperature to detect device tampering. Changes in these conditions may indicate a tampering event. You can choose an appropriate response to a detected event. Possible responses include but are not limited to the following actions:

- Device reset
- Device reset with configuration data zeroization
- BBRAM AES key destruction

You enable anti-tampering features during the design process. The configuration bitstream includes the resulting data.

Related Information

Intel Stratix 10 Device Feature Status
For more information about the status of planned Intel Stratix 10 device security features.

1.4.3. Black Key Provisioning

AES encryption helps protect confidential information or sensitive data in a configuration bitstream. When you enable AES encryption you must protect the AES key during programming, or provisioning, the AES key to the device. Typically, AES key provisioning occurs at a trusted facility at increased cost.

Black key provisioning creates a direct secure channel between your hardware security module (HSM) and the Intel Stratix 10 device. This secure channel ensures that your HSM can provision the AES key and other confidential information without exposure to an intermediate party. Black key provisioning can reduce or eliminate the need to program the AES key at a trusted facility.

Related Information

Intel Stratix 10 Device Feature Status
For more information about the status of planned Intel Stratix 10 device security features.

2. Design Authentication

FPGA designs may exhibit unintended behavior if an unauthorized client modifies the configuration bitstream. Intel Stratix 10 FPGAs include a feature to authenticate the bitstream, which helps to ensure that the bitstream is from a trusted source. Authentication uses ECDSA signatures to validate the content of a bitstream. Authentication helps to prevent the Intel Stratix 10 FPGA from configuring with an unauthorized configuration bitstream.

When you use authentication, your manufacturing process programs the hash digest of the ECDSA root public key into FPGA eFuses. The configuration bitstream contains the full root public key. The SDM computes the hash digest of the root public key and compares the computed hash digest to the hash digest stored in eFuses. The SDM only proceeds to authenticate the bitstream if the values match.

Intel Stratix 10 devices support 256- or 384-bit key length for authentication. Intel strongly recommends that you use 384-bit authentication of all new designs. If you select 384-bit authentication, the Intel Stratix 10 device uses SHA-384 with ECDSA secp384r1. If you select 256-bit authentication, the Intel Stratix 10 device uses uses SHA-256 with ECDSA prime256v1. You cannot change the root key or the authentication key length after you program the eFuses. Choose 256-bit authentication only if you have legacy hardware, such as an HSM, that cannot handle 384 bit keys.

SHA-384 generates a bitstream that is larger than SHA-256. SHA-384 hashes result in longer configuration times.

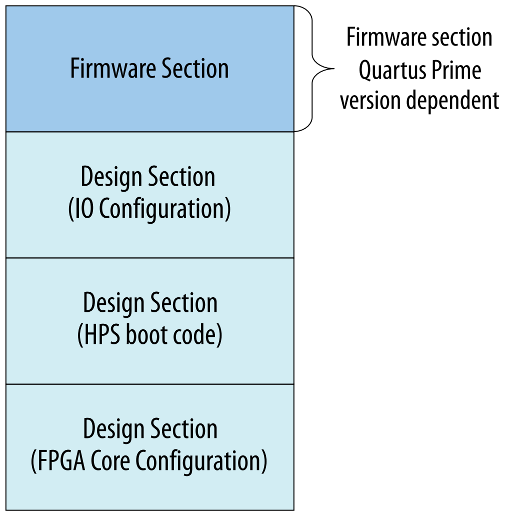
2.1. The Configuration Bitstream

The figure below shows an Intel Stratix 10 configuration bitstream that includes an FPGA and HPS. The firmware implements many functions including the functions listed here:

- FPGA configuration
- Voltage regulator configuration
- Temperature measurements
- HPS software load
- HPS reset
- Read, erase, and program flash memory
- Device security, including authentication and encryption

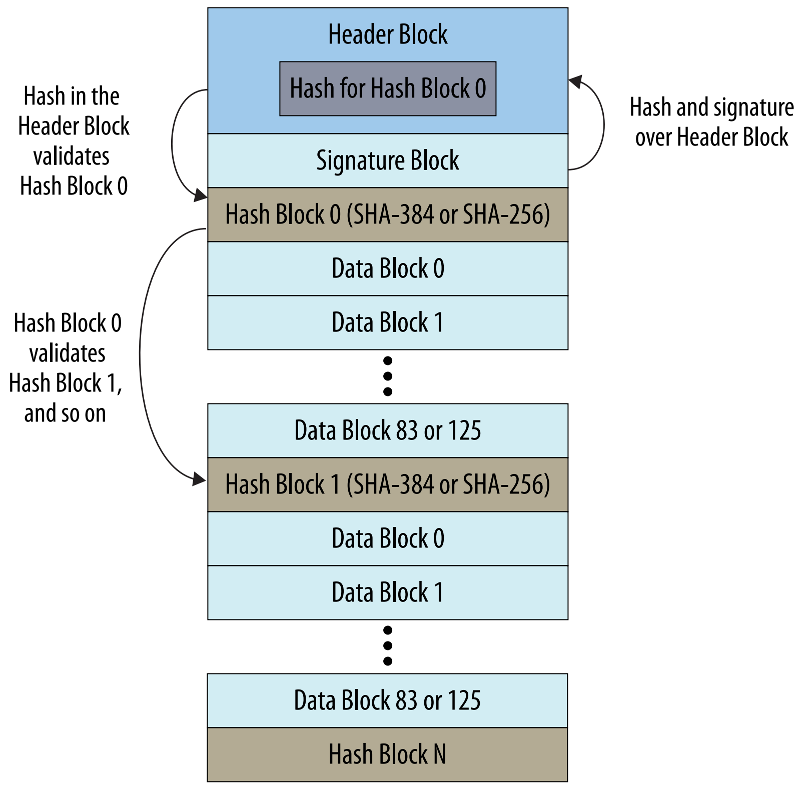
The SDM always authenticates the firmware section of the configuration bitstream. The SDM authenticates the SDM firmware section using an Intel keychain. You may also choose to sign the SDM firmware by programming the Co-signed Firmware eFuse on the device. When you enable co-signed firmware you must co-sign the firmware before generating bitstreams. The SDM validates both the Intel signature and your signature before loading and running the SDM firmware.

Figure 4. Example of an Intel Stratix 10 Configuration Bitstream Structure


Firmware Section
Design Section (IO Configuration)
Design Section (HPS boot code)
Design Section (FPGA Core Configuration)
Firmware section Quartus Prime version dependent

The I/O, HPS, and FPGA sections are dynamic and contain the device configuration information based on your design. Each dynamic section of the configuration bitstream stores information in the same format. Each section begins with a 4 kilobyte (KB) header block, followed by a signature block, hash blocks, and data.

Figure 5. Configuration Bitstream Layout


Header Block
Hash for Hash Block 0
Hash in the Header Block validates Hash Block 0
Hash and signature over Header Block
Signature Block
Hash Block 0 (SHA-384 or SHA-256)
Data Block 0
Data Block 1
Hash Block 0 validates Hash Block 1, and so on
Data Block 83 or 125
Hash Block 1 (SHA-384 or SHA-256)
Data Block 0
Data Block 1
Data Block 83 or 125
Hash Block N

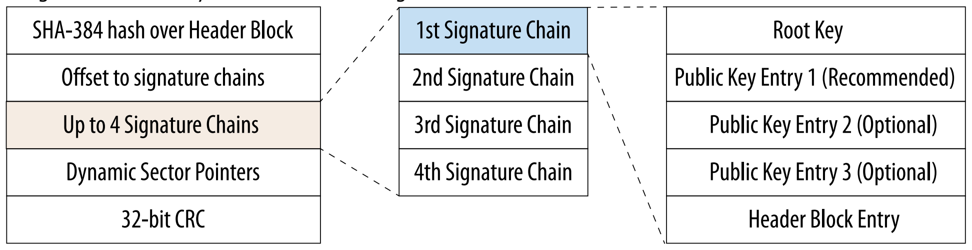
The header block contains a hash which validates hash block 0. Each hash block contains up to 125 SHA-256 hashes or 83 SHA-384 hashes. These hashes validate subsequent data blocks. A modification to any part of a section invalidates the signature. The modification results in configuration failure before the SDM processes the modified data.

2.2. Signature Block

The signature block validates the contents of the header block. After successfully validating the signatures, the SDM processes the data based on the signatures provided.

Figure 6. Signature Block Format

In this figure the Root Key is the same in all signature chains.



For more information about how the `quartus_sign` command appends the public keys to the root key to create a signature chain refer to Figure 8 on page 18.

Note: The Intel Quartus Prime Pro Edition Software GUI only supports one signature chain. You can use the `quartus_sign` command to create multiple signature chains for a Raw Binary File `.rbf`.

Table 1. Signature Block

Block	Description
SHA-384 hash of header block	This hash function checks for accidental changes in the preceding block of the configuration bitstream, typically the header block.
Signature chains	Zero or more signature chains. Each signature chain can include up to 4 keys, including the owner public root key. You can assign the other 3 keys reduced permissions so that the keys can only sign a specific section of the configuration bitstream. The Intel Quartus Prime Software supports 2 keychains for firmware signing and up to 4 keychains for the configuration bitstream. Multiple keychains provide some flexibility.
Dynamic sector pointers	Locate the design sections for the remainder of the image when you store the image in flash memory.
32-bit CRC	Protects the block from accidental modification. The CRC does not provide security. Software tools can check the CRC to identify accidental modifications.

Signature Chain Details

Intel Stratix 10 FPGAs support up to four signature chains. If a signature chain is invalid, it is ignored. The FPGA starts validating the next signature chain. To pass authentication, at least one signature keychain must pass.

Table 2. Signature Chain Content

Content	Description
Root Key Entry	The Root Entry anchors the chain to a root key known to the device. The SDM calculates the hash of the root entry and checks if the it matches the expected hash. You store the root key hash in eFuses.
Public Key Entry	Signature chains enable flexible key management. Intel recommends one public key entry in each signature chain. The previous public key signs the new public key. The public key entry provides following capabilities: • Key permission bit field to limit the sections of the configuration bitstream a public key entry can sign. The bits grant permissions for a public signing key: — Bit 0: Firmware — Bit 1: FPGA I/O, core and PR sections — Bit 2: HPS I/O and first stage bootloader (FSBL) sections — Bit 3: HPS debug certificate • For the `quartus_sign` command, specify these permissions as the equivalent hexadecimal value, 0x1, 0x2, 0x4, or 0x8. If more than one bit field is on, the key can sign more than one type of section. For example, if both bits 1 and 2 are on the permission value is 0x6 and the key can sign the FPGA I/O, core, PR, HPS I/O, and FSBL sections of the design. • Cancellation ID: Specifies the number that cancels a key that is no longer valid. Intel Stratix 10 devices support 32 cancellation IDs. Cancellation IDs 0-31 cancel owner keys. Once you cancel a key, any previous designs signed by the canceled key are unusable. You can use this feature to prevent older designs from running on a device or as part of recovery from a compromised key. Refer to Understanding Permissions and Cancellation IDs on page 15 for more information about how to manage cancellation IDs. Second- or third-level keys typically sign data. Intel Stratix 10 devices support signature chains containing up to 4 keys, including up to 3 public key entries.
Header Block Entry	The final entry in a signature chain signs the actual data. The Header Block Entry authenticates the first block of the section, and thus authenticates the whole section.

Understanding Permissions and Cancellation IDs

You use permissions to specify the types of sections that a key can sign. You can use the same or different keys for different sections. When you create a key you assign it permissions and a cancellation ID which is an integer in the range -1-31. Cancellation ID -1 is for an uncancellable key. Uncancellable keys are useful as second- or third-level keys. You can use this key to for two purposes:

- To sign other keys with the same or fewer permissions
- To sign sections directly

If you use the same cancellation ID for more than one key, canceling any key with that cancellation ID cancels all keys using that cancellation ID. For example, if you assign the same cancellation ID to both the FPGA and HPS keys, canceling the HPS key also invalidates the FPGA key. You can revalidate subsequent uncanceled keys with a signature from another key.

You cannot cancel the root key. Consequently, the root key does not have a cancellation ID. However, you can cancel a signature chain that includes two or more signature levels. Intel strongly recommends that you create a signature chain with at least two levels to retain the ability to update your signature keychain.

A good signature chain includes the following components:

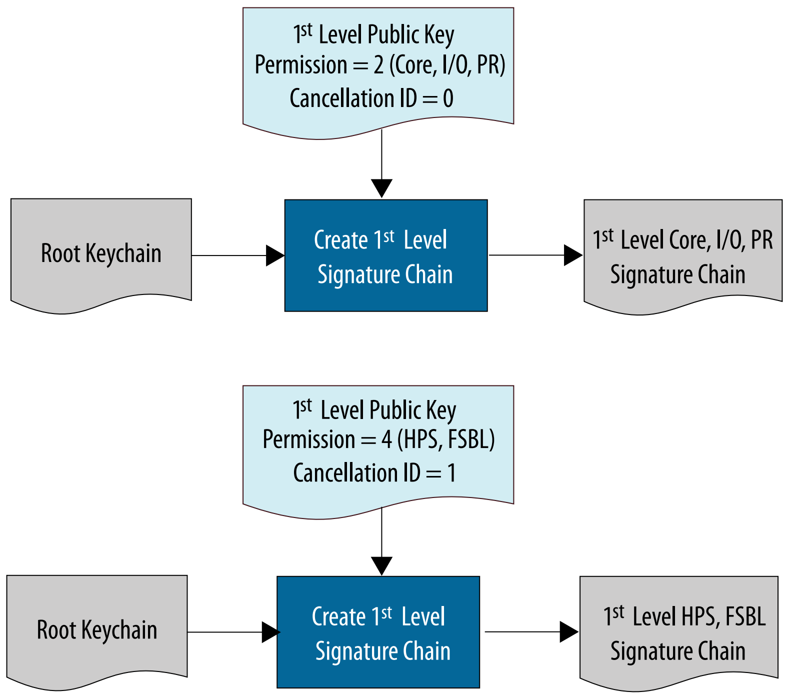
- Root key which is not cancellable on Intel Stratix 10 devices.
- First-level public key with a cancellation ID and restricted permissions.
- Optional second- and third-level public keys. Normally, these keys are not cancellable and have same permissions as the first-level key which signed them. If you can cancel one key in a key chain you can conserve cancellation IDs by using keys that are not cancellable for the optional second- and third-level keys.

Here are some reasons that you may need to cancel a signature key:

- A private key is accidentally released.
- You find a vulnerability in your design.
- You find a bug in the design after having created the signed configuration bitstream.
- You want to update the current design as part of a normal release cycle.

The Programmer performs a logical AND to determine which sections of a design a key can sign. Consequently, to create separate permissions for Core, I/O and PR logic and the HPS and FSBL, you must create two first-level keychains as shown in the following figure.

Figure 7. Create Separate Signature Chains for Different Permissions


1st Level Public Key
Permission = 2 (Core, I/O, PR)
Cancellation ID = 0
Root Keychain
Create 1st Level Signature Chain
1st Level Core, I/O, PR Signature Chain
1st Level Public Key
Permission = 4 (HPS, FSBL)
Cancellation ID = 1
Root Keychain
Create 1st Level Signature Chain
1st Level HPS, FSBL Signature Chain

2.2.1. Canceling Intel Firmware ID

If you are using device security features, Intel recommends that you update your configuration firmware to the latest available release. Additionally, Intel recommends canceling the cancellation of IDs for older versions of firmware to help ensure the device can only loads the most current firmware. This section describes when and how Intel firmware IDs are canceled.

As of Intel Quartus Prime Pro Edition Version 19.3, Intel has used the following firmware IDs.

Table 3. Intel Firmware IDs

Firmware ID	Firm Release
0-3	Early versions of firmware
4	Intel Quartus Prime Pro Edition 19.1 and 19.2
5	Intel Quartus Prime Pro Edition 19.3

When you program the owner root public key hash into a device the firmware also cancels ID eFuses to prevent older firmware from running. For example, if you use the 19.3 firmware to program the public key hash, this firmware automatically cancels IDs 0 to 4. The only situation where firmware automatically programs cancellation eFuses is during owner public key hash programming. In all other circumstances you must use the Intel Quartus Prime Programmer or mailbox commands to program eFuses.

After you have upgraded to a new version of the firmware you should prevent older versions of firmware from running by following these steps:

1. Upgrade all bitstreams stored in flash to use the new firmware version. You do not need to recompile your designs. You can recreate them by using the new version of Programmer or `quartus_pfg` to convert the `.sof` into a programming file such as `.rbf` or Programmer Object File `.pof`. You can then program the upgraded firmware into flash memory.
2. If using RSU, follow the instructions in the *Updates with the Factory Update Image* topic in the *Intel Stratix 10 Configuration User Guide* to upgrade the decision firmware and factory images in the system to the latest version. The RSU upgrade procedure protects itself against disruptions such as power failure which could interrupt the upgrade.
3. Send commands to the device to tell it to cancel the old Intel cancellation eFuses. You can use the Intel Quartus Prime Pro Edition Programmer to accomplish this task.

The firmware does not automatically program cancellation eFuses in any case except programming the root public key hash. Consequently, you can upgrade the images in flash memory before programming the cancellation eFuses.

Intel recommends adopting the following practices:

- Use the newest available firmware in your configuration bitstreams.
- Program cancellation eFuses to prevent older firmware from running on the device.

Related Information

- PCNs, PDNs, and Advisories
  For a listing of Advisories for Intel FPGAs and Programmable Devices.
- Updates with the Factory Update Image
  For the steps to update flash memory with a new factory image and the associated decision firmware and decision firmware data.

2.2.2. Authentication for HPS Software

If you are using an SoC device, the HPS Boot Code is part of the bitstream that is authenticated by the SDM during configuration.

After you successfully load the HPS Boot Code on the Intel Stratix 10 device, you may need to ensure that the following boot stages of the HPS Software are also authenticated.

The `Rocketboards` web page includes an example that uses U-boot to authenticate the subsequent boot stages of the HPS software.

Related Information

Intel Stratix 10 SoC Secure Boot Demo Design

3. Using the Authentication Feature

To authenticate an Intel Stratix 10 FPGA configuration bitstream, you prepare an authentication signature chain which includes root and public keys.

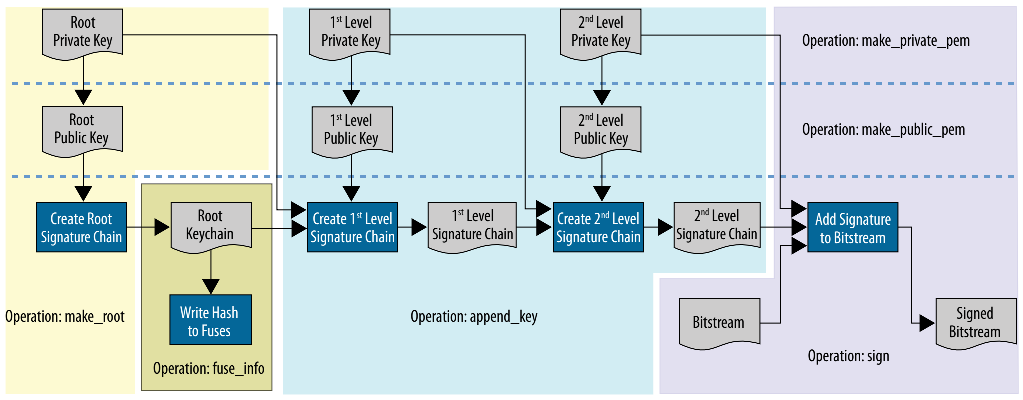
Starting with version 18.1 of the Intel Quartus Prime software, you can use the `quartus_sign` command to create a signature chain.

The following figure provides an overview of the steps to create an authentication signature chain. It shows the steps for the following operations:

1. `make_root` (light yellow)
2. `fuse_info` (darker yellow)
3. `append_key` (light blue)
4. `sign` (light gray)

The `make_private_pem` and `make_public_pem` (top right of figure) prepare the public and private keys that are inputs to the four operations listed above.

Figure 8. Steps to Create a Signature Chain



3.1. Step 1: Creating the Root Key

The root key includes public and private components. These keys are in the Privacy Enhanced Mail Certificate (PEM) format and have the `.pem` extension.

Complete the following steps to generate the root private and public keys:

1. Bring up a Nios® II command shell.

Option	Description
Windows	On the Start menu, point to **Programs** ➤ **Intel FPGA** ➤ **Nios II EDS** ➤ ***<version>*** and click **Nios II *<version>* Command Shell**.
Linux	In a command shell change to the `<install_dir>/nios2eds` and run the following command: `./nios2_command_shell.sh`

2. In the Nios II command shell, change to the directory that includes your `.sof` file.
3. Run the following command to create the private key which you use to generate the root public key.

   Note: You can create the private key with or without passphrase protection. The passphrase encrypts the private key. Intel recommends following industry best practices to use a strong, random passphrase on all private key files. Intel also recommends changing the permissions on the private `.pem` file to read-only for the owner.

Option	Description
With passphrase	`quartus_sign --family=stratix10 --operation=make_private_pem --curve=<prime256v1 or secp384r1> <root_private.pem>` Enter the passphrase when prompted to do so.
Without passphrase	`quartus_sign --family=stratix10 --operation=make_private_pem --curve=<prime256v1 or secp384r1> --no_passphrase <root_private.pem>`

4. Run the following command to create the root public key. The `root_private.pem` you generated in the previous step is an input to this command. You do not need to protect the root public key.

```
quartus_sign --family=stratix10 --operation=make_public_pem
<root_private.pem> <root_public.pem>
```

5. Convert the root public key to the Intel Quartus Prime key file format (`.qky`). You use the Intel Quartus Prime Programmer or the `quartus_pgm` command to program the root public key into a Intel Stratix 10 device. The `.qky` file is a few hundred bytes in size.

```
quartus_sign --family=stratix10 --operation=make_root <root public.pem>
<root_public.qky>
```

3.2. Step 2: Creating the Design Signing Key

You may need one or more design signing keys. You can create separate signing keys for the HPS and FPGA in Intel Stratix 10 SX devices. Creating multiple keys gives you the flexibility to cancel keys if you detect an error, uncover a vulnerability, or need to update the design.

1. Run the following command to create the first design signature private key. You use the design signature private key to create the design signature public key.

   Note: Intel recommends following industry best practices to use a strong, random passphrase on all private key files. The `curve` argument in this command must be the same has the one you specified for the root key.

Option	Description
With passphrase	`quartus_sign --family=stratix10 --operation=make_private_pem --curve=<prime256v1 or secp384r1> <design0_sign_private.pem>` Enter the passphrase when prompted to do so.
Without passphrase	`quartus_sign --family=stratix10 --operation=make_private_pem --curve=<prime256v1 or secp384r1> --no_passphrase <design0_sign_private.pem>`

2. Run the following command to create the design signature public key.

   ```
   quartus_sign --family=stratix10 --operation=make_public_pem
   <design0_sign_private.pem> <design0_sign_public.pem>
   ```

   Enter your passphrase when prompted to do so.

3.3. Step 3: Appending the Design Signature Key to the Signature Chain

This step appends design signing keys to the signature chain. The append command implements the following operations:

- Appends the 1st Level Public Key (`design0_sign_public.pem`) to the Root Public Key (`root_public.qky`) and generates the 1st Level Signature Chain (`design0_sign_public.qky`) that includes the root public key and design0 public key.
- Signs the new 1st Level Signature Chain (`design0_sign_chain.qky`) using the Root Private Key (`root_private.pem`).

1. Run the following command to append the first design signature key to the root key, creating a two-level signature chain:

   Setting the `permission` argument to 6 creates a signature that can sign the FPGA I/O, core, PR, and HPS sections. Setting the `permission` argument to 2 or 4 creates a signature that can sign only FPGA or HPS sections, respectively. Setting the `cancellation` argument to 0 means that eFuse0 can cancel this signature. eFuses 0-31 are available for owner cancellation.

   ```
   quartus_sign --family=stratix10 --operation=append_key \
    --previous_pem=<root_private.pem> --previous_qky=<root_public.qky> \
    --permission=6 --cancel=0 <design0_sign_public.pem> \
    <design0_sign_chain.qky>
   ```

2. Use `append_key` again to create a three-level signature chain:

   a. Repeat the commands in Step 1 on page 20, to generate both `design1_sign_private.pem` and `design1_sign_public.pem`.

   b. Append `design1_sign_public.pem` to the signature chain.

Setting the cancellation argument to 1, means that the second available cancellation eFuse, eFuse 1, cancels this signature.

```
quartus_sign --family=stratix10 --operation=append_key \
 --previous_pem=<design0_sign_private.pem> \ --
previous_qky=<design0_sign_chain.qky> --permission=6 \
 --cancel=1 <design1_sign_public.pem> <design1_sign_chain.qky>
```

Enter the passphrase when prompted to do so.

3. If you are generating separate keychains for HPS and FPGA signing, repeat steps 1 and 2 with different PEM files. The FPGA signing chain should have `permission=2`. The HPS signing chain should have `permission=4`.

3.4. Step 4: Signing the Bitstream

Once you generate the private PEM and `.qky` files, you are ready to sign the bitstream. There are two options for bitstream signing:

- You use Intel Quartus Prime Programming File Generator to generate the signed bitstream from a `.sof` file. You specify the required format for your configuration scheme. The JTAG Indirect Configuration File (`.jic`) and Raw Programming Data File (`.rpd`) formats are available for Active Serial (AS) configuration. The Programmer Object File `.pof` and `.rbf` are available for Avalon® Streaming (Avalon-ST) configuration.
- Alternatively, you can use `quartus_sign` command to sign the bitstream. This command requires the `.rbf` as the input to generate a signed `.rbf` file.

Note: If you are using the Jam* Standard Test and Programming Language (STAPL) Player to program over JTAG the following command converts an `.rbf file` to the `.jam` format that the Jam STAPL Player requires:

```
quartus_pfg -c signed_bitstream.rbf signed_bitstream.jam
```

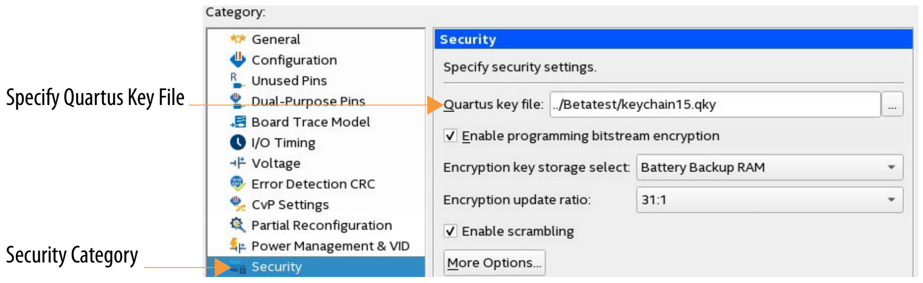
3.5. Step 4a: Signing the Bitstream Using the Programming File Generator

The Programming File Generator requires the private key file (`.pem`) to sign the configuration bitstream. You append the generated signature chain (`.qky`) to your compiled design `.sof`. Attaching the signature chain to your `.sof` does not require you to recompile your design.

Complete the following steps to append the signature chain key file to the `.sof` file and generate the signed bitstream using the Programming File Generator.

1. Choose one of the following options to append the signature chain key file the configuration bitstream:
   — Specify the `.qky` file using the Intel Quartus Prime software. On the Assignment tab, select **Device ➤ Device and Pin Options ➤ Security ➤ Quartus Key File**. Then browse to your signature key chain file.

Figure 9. Specifying the Quartus Key File



Security

Specify security settings.

Quartus key file: ../Betatest/keychain15.qky

☑ Enable programming bitstream encryption

Encryption key storage select: Battery Backup RAM

Encryption update ratio: 31:1

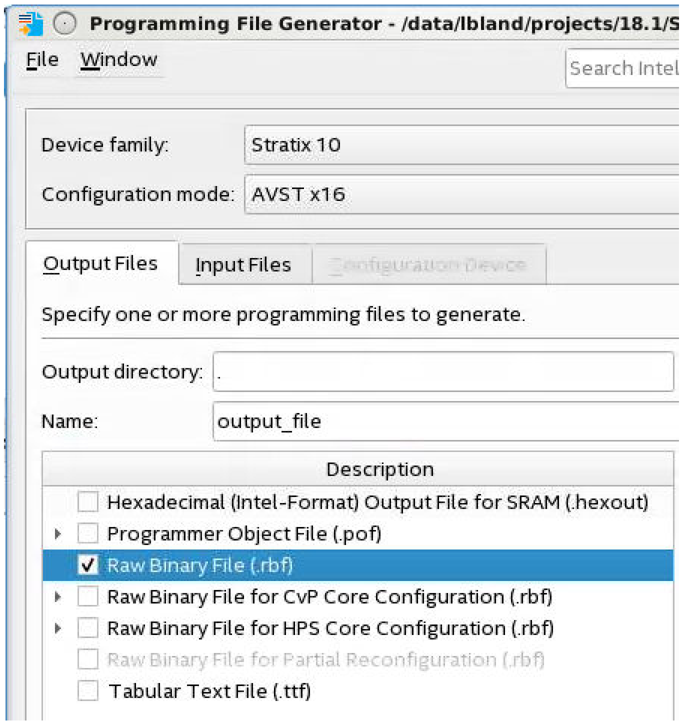
☑ Enable scrambling

— Alternatively, you can add the following assignment statement to your Intel Quartus Prime Settings File (`.qsf`):

```
set_global_assignment -name QKY_FILE design1_sign_keychain.qky
```

2. To generate a `.sof` that includes `design1_sign_keychain.qky` select **Processing ➤ Start ➤ Start Assembler**.
   The new `.sof` includes the `design1_sign_keychain.qky` signature chain.
3. On the Intel Quartus Prime file menu, select **File ➤ Programming File Generator.**

Figure 10. Programming File Generator



Device family: Stratix 10

Configuration mode: AVST x16

Specify one or more programming files to generate.

Output directory: .

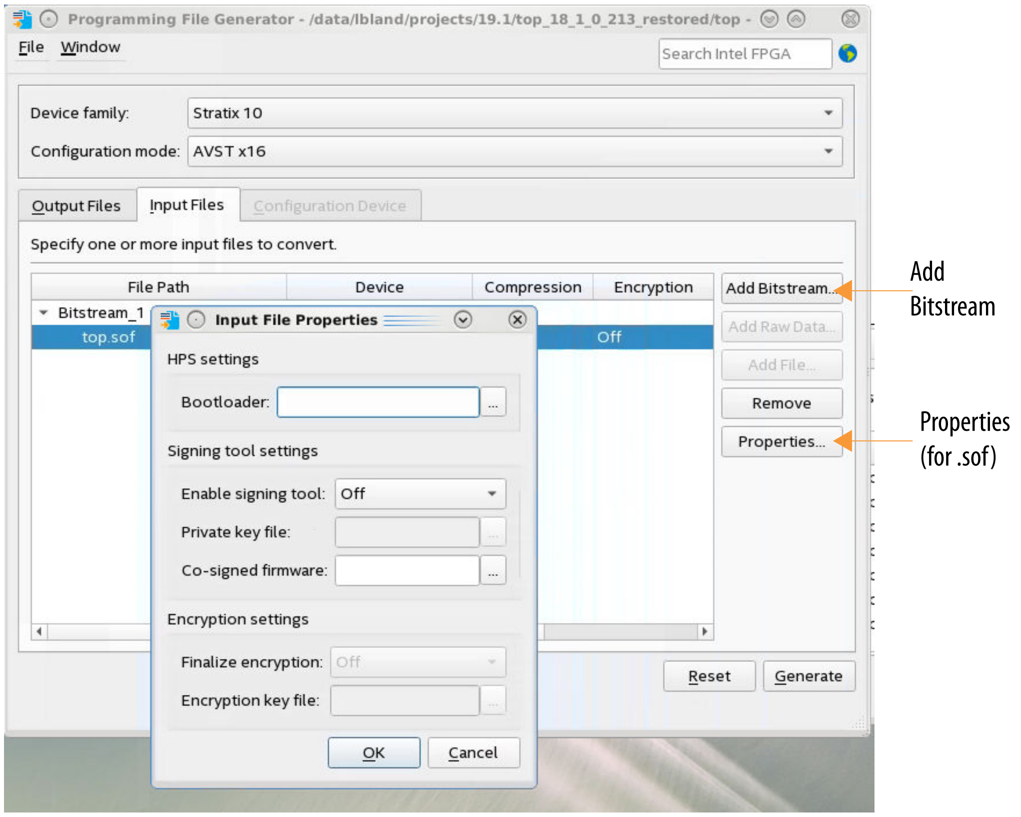
Name: output_file

☐ Hexadecimal (Intel-Format) Output File for SRAM (.hexout)
☐ Programmer Object File (.pof)
☑ Raw Binary File (.rbf)
☐ Raw Binary File for CvP Core Configuration (.rbf)
☐ Raw Binary File for HPS Core Configuration (.rbf)
☐ Raw Binary File for Partial Reconfiguration (.rbf)
☐ Tabular Text File (.ttf)

4. For **Device family**, select **Intel Stratix 10**
5. For **Configuration mode**, select the configuration mode you plan to use. This example uses **AVST x16**.
6. For **Output directory** click **Browse** and navigate to your output files directory.

7. On the Output Files tab, select **Raw Binary File (.rbf)**.
8. On the Input Files tab, click **Add Bitstream** then browse and select your `.sof` file.

Figure 11. Input File Properties



Device family: Stratix 10
Configuration mode: AVST x16

Specify one or more input files to convert.

HPS settings

Bootloader:

Signing tool settings

Enable signing tool: Off
Private key file:
Co-signed firmware:

Encryption settings

Finalize encryption: Off
Encryption key file:

9. On the Input Files tab, click **Add Bitstream** and then browse to your bitstream.
10. On the **Input Files** tab, click **Properties...** and make the following selections under **Signing tool settings**:
    a. Select **On** for **Enable signing tool**.
    b. For **Private key file**, select the final private signing key. For example, Figure 6 Figure 8 on page 18 *Steps to Create a Signature Chain* shows a root private key and two private keys. For this key chain, you would select the second-level private `.pem` file.

       Note: If your `.pem` is password-protected, the GUI opens a dialog box to enter the password.

3.6. Step 4b: Signing the Bitstream Using the quartus_sign Command

The `quartus_sign` command takes the signature chain (`.qky`), a private signing key (`.pem`), and the unsigned raw binary file (`.rbf`) as inputs to generate the signed `.rbf`.

You can generate the unsigned bitstream in `.rbf` format using the following command:

```
quartus_pfg -c design.sof unsigned_bitstream.rbf
```

1. Run the following command to sign the bitstream using a command-line command:

   ```
   > quartus_sign --family=stratix10 --operation=sign \
   --qky=design1_sign_keychain.qky --pem=design1_sign_private.pem \
    unsigned_bitstream.rbf signed_bitstream.rbf
   ```

Related Information

Generating Secondary Programming Files with Programming File Generator

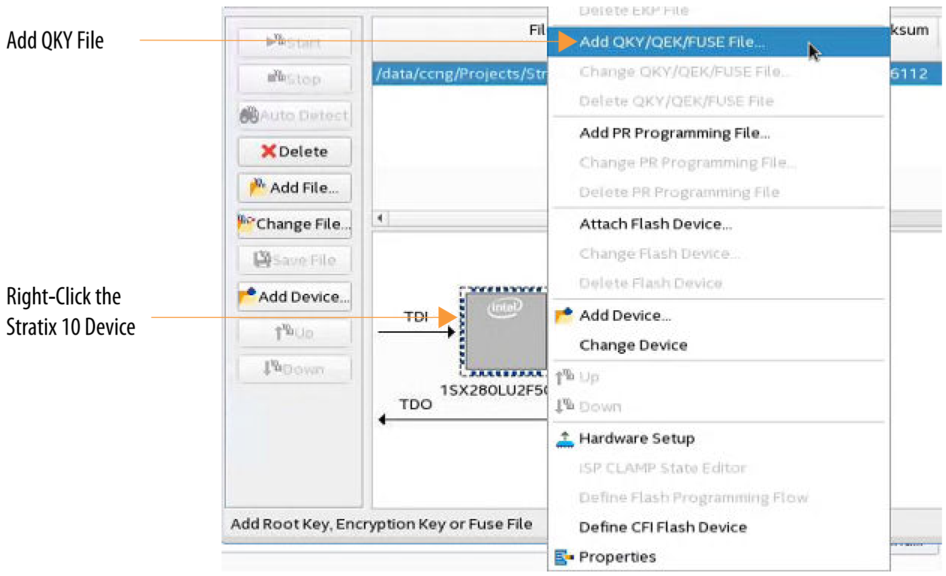
3.7. Step 5: Programming the Owner Root Public Key for Authentication

Your manufacturing process programs the hash of the owner root public key, `root_public.qky`, into eFuses available on the Intel Stratix 10 device. Programming the hash value into actual eFuses on the device is irreversible. During development, you can validate the hash value by programming this value into virtual eFuses. The virtual eFuses are volatile. Values stored in eFuses clear each time you power cycle the Intel Stratix 10 device.

You can use the Intel Quartus Prime Software to program the public root key for authentication. Alternatively, you can use a command-line command to accomplish this task.

3.8. Step 5a: Programming the Owner Root Public Key

1. On the Tools menu, select **Programmer**.
2. Right click the image of the Intel Stratix 10 device and select **Edit ➤ Add QKY/QEK/Fuse file ...**.



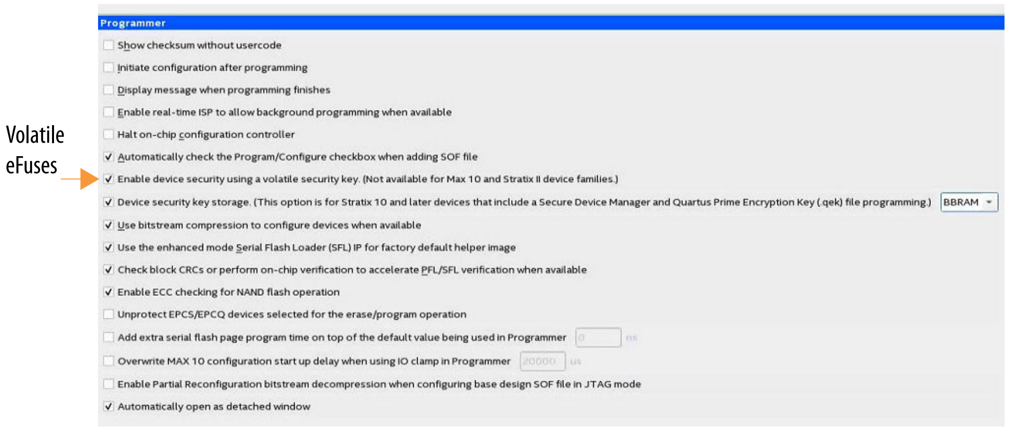
3. Browse to the owner root public key file and click **Open**.

   Note: Once you have specified the QKY file, the programmer displays the compatible version of firmware that you use to program the device. The version of the Intel Quartus Prime Programmer and the firmware must match.

4. You can choose to program the non-volatile eFuses or simulate the actual hardware using virtual eFuses.

   Caution: Incorrect fuse programming can make your device unusable. Intel recommends that you test all eFuse programming sequences using virtual fusing before you program physical eFuses on your first device.

   — To select virtual eFuses, on the Programmer Tools menu, select **Options**. Turn on **Enable device security using a volatile security key** if this option is not already on. By default this option is on. Then, select **OK**.

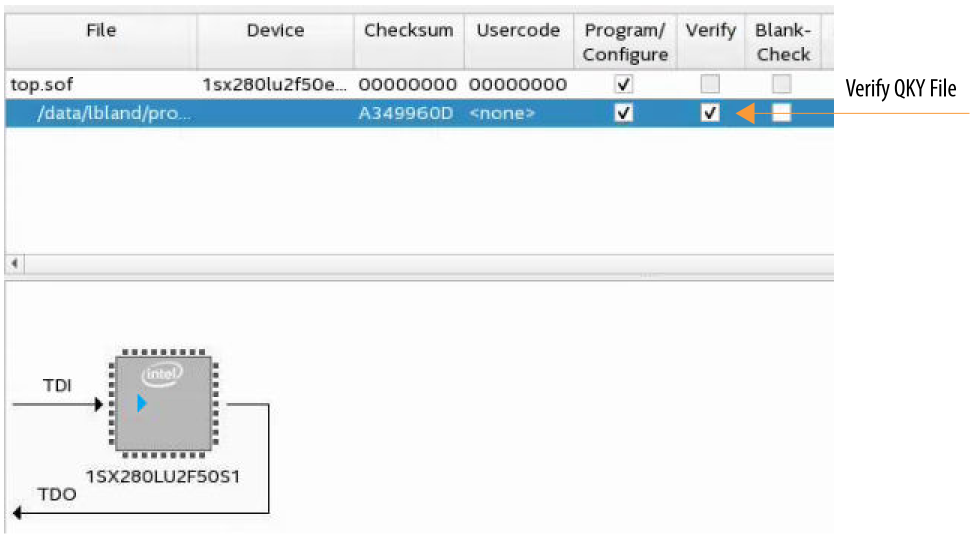


Programmer

- [ ] Show checksum without usercode
- [ ] Initiate configuration after programming
- [ ] Display message when programming finishes
- [ ] Enable real-time ISP to allow background programming when available
- [ ] Halt on-chip configuration controller
- [✓] Automatically check the Program/Configure checkbox when adding SOF file
- [✓] Enable device security using a volatile security key. (Not available for Max 10 and Stratix II device families.)
- [✓] Device security key storage. (This option is for Stratix 10 and later devices that include a Secure Device Manager and Quartus Prime Encryption Key (.qek) file programming.) BBRAM
- [✓] Use bitstream compression to configure devices when available
- [✓] Use the enhanced mode Serial Flash Loader (SFL) IP for factory default helper image
- [✓] Check block CRCs or perform on-chip verification to accelerate PFL/SFL verification when available
- [✓] Enable ECC checking for NAND flash operation
- [ ] Unprotect EPCS/EPCQ devices selected for the erase/program operation
- [ ] Add extra serial flash page program time on top of the default value being used in Programmer 0 ns
- [ ] Overwrite MAX 10 configuration start up delay when using IO clamp in Programmer 20000 us
- [ ] Enable Partial Reconfiguration bitstream decompression when configuring base design SOF file in JTAG mode
- [✓] Automatically open as detached window

— To select the actual non-volatile eFuses, on the Programmer Tools menu, select **Options**. Turn off the **Enable device security using a volatile security key** option.

5. To verify that the fuse value and the hash value of the owner root public key match, turn on the **Verify** option in the Intel Quartus Prime software.



File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check
top.sof	1sx280lu2f50e...	00000000	00000000	✓		
/data/lbland/pro...		A349960D	<none>	✓	✓	

3.9. Step 5b: Calculating the Owner Root Public Key Hash

1. Use the `quartus_sign` command with the operation set to the `fuse_info` operation to generate the hash of the root public key, as follows:

```
quartus_sign --family=stratix10 --operation=fuse_info \
 public_root.qky hash_fuse.txt
```

To validate the owner root public key hash, you can compare the value of `hash_fuse.txt` to the value you observe when turn on the **Examine** option while configuring the Intel Stratix 10 device in the Intel Quartus Prime Pro Edition Programmer.

Related Information

Using eFuses on page 48

4. Co-Signing Device Firmware Overview

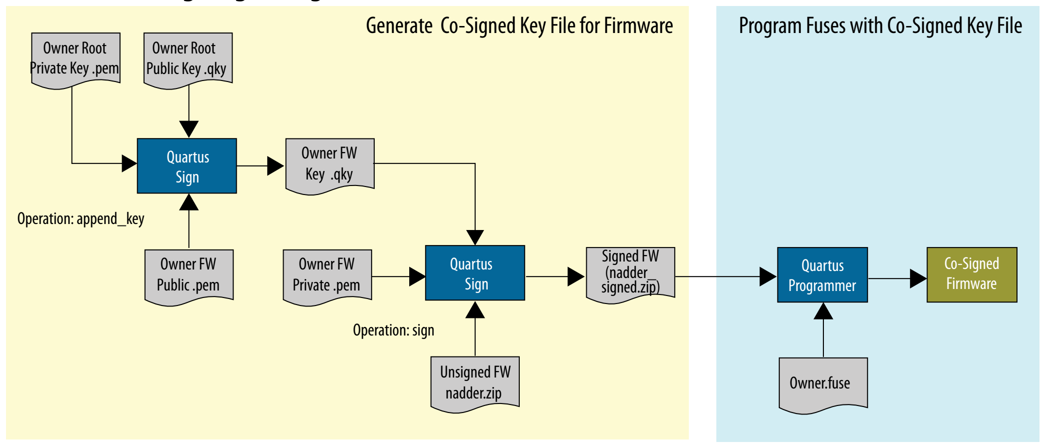
Intel programs each Intel Stratix 10 device with an Intel root public key hash during the manufacturing process. Boot code stored in read-only memory in the Intel Stratix 10 SDM uses this hash to validate an Intel signature chain. This process helps to ensure that only firmware that Intel has approved can run on the device. Intel only signs firmware after a rigorous audit process.

The Intel Quartus Prime Software supports co-signing device firmware. Co-signing adds another layer of protection for device firmware. The joint signature capability allows you to sign device firmware with an owner signing key that you generate. You enable the co-signature by programming the owner root public key hash and the co-signed firmware eFuses. Once you program these security fuses, loading new firmware requires both Intel and owner signatures.

4.1. Using the Co-Signing Feature

The following figure provides an overview of the steps to create an signature chain to co-sign the device firmware.

Firmware Co-Signing Design Flow



It shows the steps for the following operations:

1. Generating an owner firmware key and appending this key owner FW public `.pem`) to the existing owner keychain (owner FW key`.qky`).
2. Co-signing the firmware. Add the owner signature to `nadder.zip` using the new keychain and the Owner FW Private `.pem` file.

Note: The `nadder.zip` file is available in the `<install_dir>/quartus/common/devinfo/programmer/firmware/` directory. This file includes the SDM firmware.

3. Programming the Co-Signed Firmware eFuses in the the Intel Stratix 10 device using the signed firmware (Signed FW `signed_nadder.zip`) and `owner.fuse` as inputs.

Note: You must power cycle your board after programming the fuses.

4.1.1. Prerequisites for Co-Signing Device Firmware

Before completing the steps to co-sign device firmware, you must generate an owner root key and program the owner root public key hash eFuse in the eFuses on your Intel Stratix 10 device.

To generate the owner root key follow the instructions in *Using the Authentication Feature Step 1: Creating the Root Key* or by using your own custom hardware security module.

Then program the owner root public key hash into eFuses. By default, the `quartus_pgm` command programs the root public key hash into virtual (volatile) eFuses. You can use the optional `--non_volatile_key` argument to specify physical eFuses on the Intel Stratix 10 device. Here are both versions of the `quartus_pgm` command: :

```
//For physical (non_volatile) eFuses on the Intel Stratix 10 device
quartus_pgm -c 1 -m jtag -o "p;root_public.qky" --non_volatile_key
```

```
//For virtual (volatile) eFuses
quartus_pgm -c 1 -m jtag -o "p;root_public.qky"
```

Alternatively, you can use the Intel Quartus Prime Programmer to program the owner root key as described in *Step 5: Programming the Owner Public Root Key for Authentication*.

Related Information

- Step 5: Programming the Owner Root Public Key for Authentication on page 24
- Step 1: Creating the Root Key on page 19

4.1.2. Generating the Owner Firmware Signing Key

You use the Intel Quartus Prime Signing Tool `operation=append_key` to append a firmware signing key to the owner root public key. The permission is set to 1 for firmware.

The first two steps generate required inputs to the `operation=append_key` command shown in Step 3.

1. Run the following command to generate a private key you use to sign the firmware.

```
quartus_sign --family=stratix10 --operation=make_private_pem --
curve=prime256v1 or <secp384r1> owner_fw_private.pem
```

Send Feedback

2. Run the following command to generate the corresponding firmware public key from `owner_fw_private.pem`.

```
quartus_sign --family=stratix10 --operation=make_public_pem
owner_fw_private.pem owner_fw_public.pem
```

3. Run the following command to append the `owner_fw_public.pem` to the owner root keychain

```
quartus_sign --family=stratix10 --operation=append_key \
--previous_pem=owner_root_private.pem --previous_qky=owner_root_public.qky
 --permission=0x1 --cancel=1 owner_fw_public.pem owner_fw_key.qky
```

4.1.3. Co-Signing the Firmware

You use the Intel Quartus Prime Signing Tool `operation=sign` to sign the firmware with your private firmware key.

1. Run the following command to co-sign the firmware file. The firmware file is `nadder.zip`. The Intel Quartus Prime Software writes this file to the `<install_dir>/ quartus/common/devinfo/programmer/firmware/` directory.

```
quartus_sign --family=stratix10 --operation=sign --qky=owner_fw_key.qky \
--pem=owner_fw_private.pem nadder.zip nadder_signed.zip
```

Refer to Programming eFuses on page 51 for instructions on programming eFuses.

4.1.4. Powering On In JTAG Mode After Implementing Co-Signed Firmware

After you program the co-signed firmware eFuse, the Intel Stratix 10 FPGA requires all configuration bitstreams to include co-signed firmware on every subsequent power-on. The existing helper image containing the SDM firmware is now out-of-date because it does not specify co-signed firmware. You must regenerate a new `signed_helper_image.rbf` file that specifies co-signed firmware.

Use the co-signed `signed_nadder_signed.zip` to regenerate the `signed_helper_image.rbf`. Load the `.rbf` then, program the `.fuse` file.

1. Generate a signed helper image for eFuse programming.

```
quartus_pfg --helper_image -o helper_device=1SG280HN2 -o subtype=FUSE \
 -o fw_source=signed_nadder.zip signed_helper_image.rbf
```

2. Configure your Intel Stratix 10 device with the `signed_helper_image.rbf` file you just created.

```
quartus_pgm -c 1 -m jtag -o “p;signed_helper_image.rbf”
```

5. HPS Debug Using a Certificate

For Intel Stratix 10 SX devices, you can require an HPS debug certificate before permitting access to the JTAG interface for HPS debugging. An HPS debug certificate is a one-time certificate that is valid until you power down the SDM or reconfigure the device. Restarting the HPS does not invalidate the HPS debug certificate.

Signing a configuration bitstream with the HPS debug access port (DAP) available without a debug certificate enables that configuration bitstream to load unauthenticated software to the HPS. In response, the Intel Quartus Prime Pro Edition Software generates critical warnings. Intel recommends careful consideration before using this option. Intel strongly recommends canceling the signing key ID after this configuration bitstream is no longer needed.

Note: You can debug the HPS without a certificate by turning on the **Allow HPS debug without certificate** on the **Assignments ➤ Device ➤ Device and Pin Options ➤ Configuration** menu.

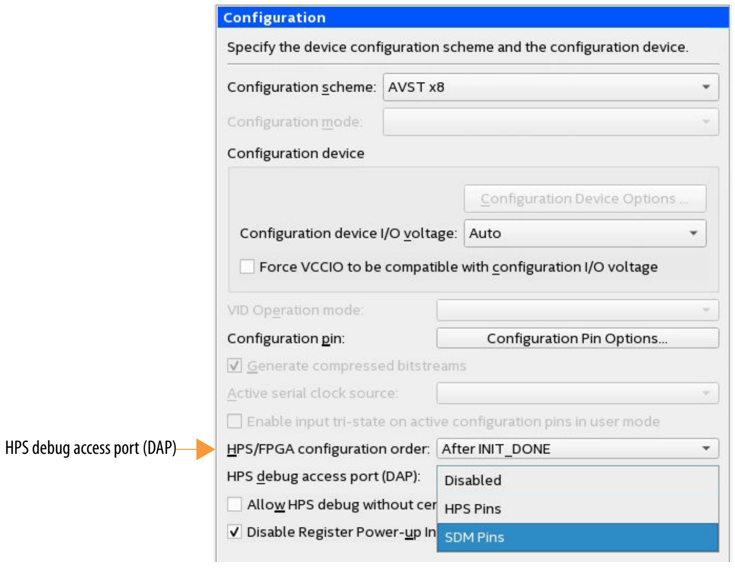
Using an HPS debug certificate includes the following steps:

1. Requesting the certificate from a configured device.
2. Signing the certificate using a keychain with HPS debug permissions.
3. Programming the signed certificate back into the device.

You can create an HPS debug certificate when the following conditions are true:

- You have selected either HPS or SDM pins to access the HPS.

Figure 12. Specify Either HPS or SDM Pins for the HPS DAP



Configuration

Specify the device configuration scheme and the configuration device.

Configuration scheme: AVST x8

Configuration mode:

Configuration device

Configuration Device Options ...

Configuration device I/O voltage: Auto

Force VCCIO to be compatible with configuration I/O voltage

VID Operation mode:

Configuration pin: Configuration Pin Options...

✓ Generate compressed bitstreams

Active serial clock source:

Enable input tri-state on active configuration pins in user mode

HPS/FPGA configuration order: After INIT_DONE

HPS debug access port (DAP): Disabled / HPS Pins / SDM Pins

Allow HPS debug without cer

✓ Disable Register Power-up In

- You have not disabled the HPS DAP.
- You have programmed the FPGA with an owner root key. Refer to *Step 5: Programming the Owner Public Root Key for Authentication*for more information.
- You have programmed the device with a signed bitstream with the HPS and FSBL permission set to true. (permission=4 for HPS and FSBL)
- You have not permanently disabled HPS debugging on the device by programming the JTAG disable eFuse. For more information about the available eFuses refer to the *Owner Programmable eFuses* table in the *Using eFuses* topic.
- You have not programmed the FPGA with a design that disables HPS debug. HPS debug certificates do not override the setting to disable HPS debug for a given bitstream.

Related Information

- Step 1: Creating the Root Key on page 19
- Using eFuses on page 48

5.1. Enabling HPS JTAG Debugging

Use this procedure to enable HPS JTAG debugging after configuring the Intel Stratix 10 SX device with a signed bitstream.

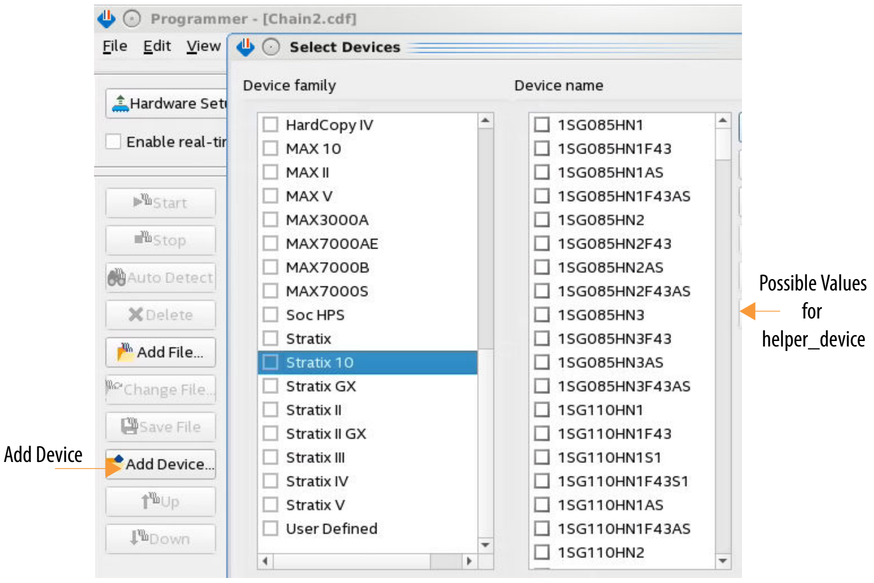
You should already have created a first-level signature chain by completing the instructions in the following topics:

1. Step 2: Creating the Design Signing Key on page 19.
2. Step 3: Appending the Design Signature Key to the Signature Chain on page 20. Be sure to specify `permission=4` for the HPS and FSBL.

Completing these commands results in <`design0_sign_chain.qky`> and <`design0_sign_private.pem`> files that are inputs the `quartus_sign` command the creates the signed HPS debug certificate.

1. To create the HPS debug certificate, you must provide a <`device`> argument to the `quartus_pgm` command. Use the Intel Quartus Prime Programmer **Device name** list to determine the proper <`device`> argument by completing the following steps:
   a. Find the list of Intel Stratix 10 devices, in the Intel Quartus Prime Programmer, by select **Add Device.**
   b. In the **Device family** list, select **Intel Stratix 10**. In the **Device name** list, find the part number that matches your device.

Figure 13. User the Programmer to Determine the helper_device Argument



2. Generate an unsigned secure HPS debug certificate from the programmed device. The <`device`> argument is the **Device name** you identified in the previous step.

```
quartus_pgm -c 1 -m jtag -o “ei;unsigned_hps_debug.cert;<device>”
```

3. Sign the HPS debug certificate using the `quartus_sign` command:

```
quartus_sign --family=stratix10 --operation=sign \
--qky=design0_sign_chain.qky --pem=design0_sign_private.pem \
unsigned_hps_debug.cert signed_hps_debug.cert
```

4. Send the signed HPS debug certificate to the device to enable HPS debugging.

```
quartus_pgm -c 1 -m jtag -o “p;signed_hps_debug.cert"
```

Send Feedback

6. Signing Command Detailed Description

The signing command, `quartus_sign`, supports the following functions:

- Generates the private and public PEM files
- Generates the signature chain starting with the root public key
- Appends additional public keys to the signature chain
- Signs a bitstream, firmware, or debug certificate
- Calculates the root public key hash from the signature chain file `.qky` file

The `quartus_sign` command always specifies the FPGA device family and operation. Here is the general format of the command:

```
quartus_sign --family=stratix10 --operation=<type of operation> [additional
arguments]
```

The following table summarizes all the `quartus_sign` operations. "Creates a new Quartus keychain .qky file with a given public key .pem in the root entry" (eg, this command does not generate a new key)

Table 4. Signing Command Argument Summary

Argument	Options	Description
`operation`	`make_private_pem`	Generates a private key in `.pem` format such as `root_private.pem`.
	`make_public_pem`	Generates a public key in `.pem` format from the private `.pem` such as `root_public.pem`.
	`make_root`	Creates a new Intel Quartus Prime keychain `.qky` file with a given public key `.pem` in the root entry such as `root_public.qky`.
	`append_key`	Signs, appends, and sets the permissions and cancellation ID of an additional public key to an existing signature chain in the Intel Quartus Prime keychain `.qky` format.
	`sign`	Signs the bitstream with the `.pem` private key and key chain.
	`fuse_info`	Calculates the root public key hash from the `.qky` file.

The following topics provide details on each operation. The operations are listed in the order that you normally run them to create a signature chain, sign the bitstream, and calculate the root public key hash.

6.1. Generate Private PEM Key

The first step in generating the signature chain is creating the private PEM.

Command	`quartus_sign --family=stratix10 --operation=make_private_pem --curve=<prime256v1 or secp384r1> <output private PEM file>`
	or:
	`quartus_sign --family=stratix10 --operation=make_private_pem --curve=<prime256v1 or secp384r1> --no_passphrase <output private PEM file>`

Input file	None
Output file	Private PEM file
Arguments	This command includes 1 required argument and 1 optional argument: • `curve`: Selects the Elliptic Curve Digital Signature Algorithm (EDCSA) 256 or 384. Intel recommends using `secp384r1` if possible because `prime256v1` may be vulnerable to attacks within the next 20 years. • `no_passphrase`: By default the `make_private_pem` command encrypts the private key. You can add the optional `--no_passphrase` argument to create a plain text key. Intel recommends following industry best practices to use a strong, random passphrase on all private key files.

6.2. Generate Public PEM Key

The second step in generating the signature chain is generating the public PEM file from the private PEM file.

Command	`quartus_sign --family=stratix10 --operation=make_public_pem <input private PEM file> <output public PEM file>`
Input file	`input private PEM file`: This is the file that the `make_private_pem` generates.
Output file	`output public PEM file`: `make_public_pem` generates this file.
Arguments	This command has no additional arguments.

6.3. Generate Root Signature Chain

The third step in generating the signature chain makes the root public key by converting the public key PEM file to the Intel Quartus Prime key format.

Command	`quartus_sign --family=stratix10 --operation=make_root <input root public PEM file> <output root public qky file>`
Input file	`input public PEM file`: This is the file that the `make_public_pem` generates.
Output file	`output root public qky file`: `make_root` generates this file.
Arguments	This command has no additional arguments.

6.4. Append Key to Signature Chain

The append command implements the following functions:

- Uses the private part of the last-appended public key to sign the new public key
- Appends the specified design signing key to the root public Intel Quartus Prime keychain
- Assigns specified permissions and cancellation ID to the appended public key

Command	`quartus_sign --family=stratix10 --operation=append_key --previous_pem==<private PEM for the public key of last entry in input QKY> --previous_qky=<input QKY> --permission=<permission value to authenticate> --cancel=<cancel ID> <public PEM for new entry> <output QKY>`
Input files	The `append_key` command has 3 input files: • `previous_pem`: The private PEM file that is input to the `make_public_pem` operation. This private PEM is from the previous entry in the input signature chain. • `previous_qky`: Intel Quartus Prime `.qky` format keychain to which the the `quartus_sign` command appends the new public key.
Output file	The `append_key` outputs 1 file: • `output_key`: This is the new signature chain with one additional entry.
Arguments	This command includes 2 arguments: • `permission`: Sets the signing key permission value. These bits are positional. Each bit grants permission to sign a particular type of data. To allow a key to sign more than one type of data, you can add the permissions for the data types. For example, a permission value of 6 can sign all data types that permissions 2 and 4 can sign. The following values are valid: — 0: To sign firmware — 2: To sign FPGA I/O, core sections, and PR sections — 4: To sign HPS I/O and the FSBL — 8: To sign an HPS debug certificate • `cancel`: Specifies the cancellation ID to cancel this signature. The valid range is 0-31. The special value of -1 is for keys that are uncancellable.

6.5. Sign the Bitstream, Firmware, or Debug Certificate

The sign operation takes an unsigned `image.rbf`, `firmware.zip`, `debug.cert` as input. The sign operation generates a signed output file, either `signed_image.rbf`, `signed_firmware.zip`, or `signed_debug.cert`.

For `.rbf` generation, you convert the `.sof` to an `.rbf` using either the Intel Quartus Prime **File ➤ Programming File Generator** dialog box or the `quartus_pfg` command-line command.

Command	`quartus_sign --family=stratix10 --operation=sign --qky=<qky file> --pem=<private PEM for the public key of last entry in the input QKY> <unsigned rbf, unsigned nadder file, unsigned debug certificate> <signed rbf, nadder file, or signed debug certificate>`
Input file	The sign operation supports the following 3 input file types: • `unsigned rbf file`: This is the `.rbf` that you generate from the `.sof` • `unsigned nadder file`: `quartus/common/devinfo/programmer/firmware/nadder.zip`. This file contains the SDM firmware. • `unsigned debug certificate file`: This is the unsigned debug certificate you create from the programmed device by running the following command: `quartus_pgm -c 1 -m jtag -o “ei;hps_unsigned.cert;<device>"`
Output file	`signed rbf file`, `signed nadder file`, or `signed certificate file`: This file is the output of the `sign` operation.
Arguments	This command has 2 additional arguments: • `qky`: This is the `.qky` file generated by the previous `append_key` or `make_root` operation. • `pem`: This is the private PEM for the previous public key of the input QKY.

Refer to Step 4: Signing the Bitstream for the steps to sign the bitstream using the Programing File Generator tool.

Related Information

Step 4: Signing the Bitstream on page 21

6.6. Calculate Root Public Key Hash from QKY

The `fuse_info` operation returns the hash of the root public key.

Command	`quartus_sign --family=stratix10 --operation=fuse_info <input QKY> <fuse output text>`
Input file	`input QKY`: This is the root public key.
Output file	`fuse output text`: Manufacturing uses this text file to program the specified eFuses of the Intel Stratix 10 device.
Arguments	This command has no additional arguments.

7. Encryption and Decryption Overview

A single AES root key under owner control encrypts the dynamic blocks of the Intel Stratix 10 configuration bitstream.

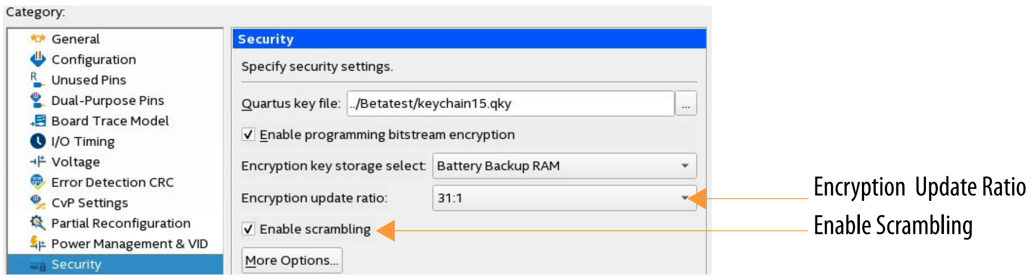
Encryption Process

You can store the owner AES Root key in virtual eFuses, physical eFuses, BBRAM, or a PUF-wrapped key. Using the PUF to wrap the AES root key for storage in flash is planned in a future release. To prevent overuse of the AES root key, the root key encrypts a chain of intermediate keys. The root key encrypts the first intermediate key, which encrypts the second intermediate key, and so on. The last intermediate key encrypts the section keys.

Encryption supports up to 20 intermediate keys. By default, the encryption function uses three intermediate keys. These keys mitigate side channel attack risk by limiting the exposure of the final key to encrypt a given section of the bitstream.

Intel recommends that you limit the amount of data encrypted with the same key using AES update mode. You enabled AES update mode by setting the **Assignments ➤ Device ➤ Device and Pin Options ➤ Security ➤ Encryption Update Ratio** parameter to **31:1**. When enabled, the Intel Quartus Prime Pro Edition Software inserts keys to limit the amount of data encrypted by a given key to the specified ratio. For example, the **31:1** ratio inserts a new key every 31 * 32 bytes. Different ratios may become available in a future release.

Figure 14. Security Tab: AES Encryption Key Update Ratio and Enable Scrambling



Security

Specify security settings.

Quartus key file: ../Betatest/keychain15.qky

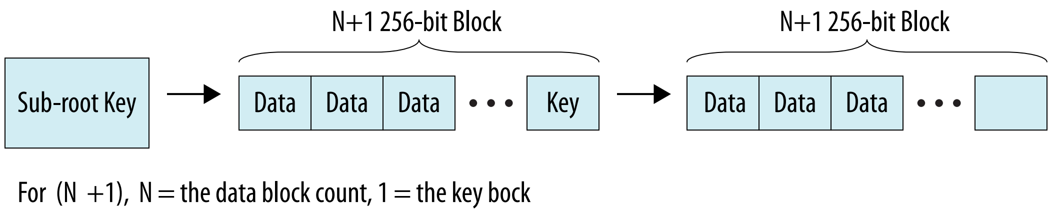
✓ Enable programming bitstream encryption

Encryption key storage select: Battery Backup RAM

Encryption update ratio: 31:1

✓ Enable scrambling

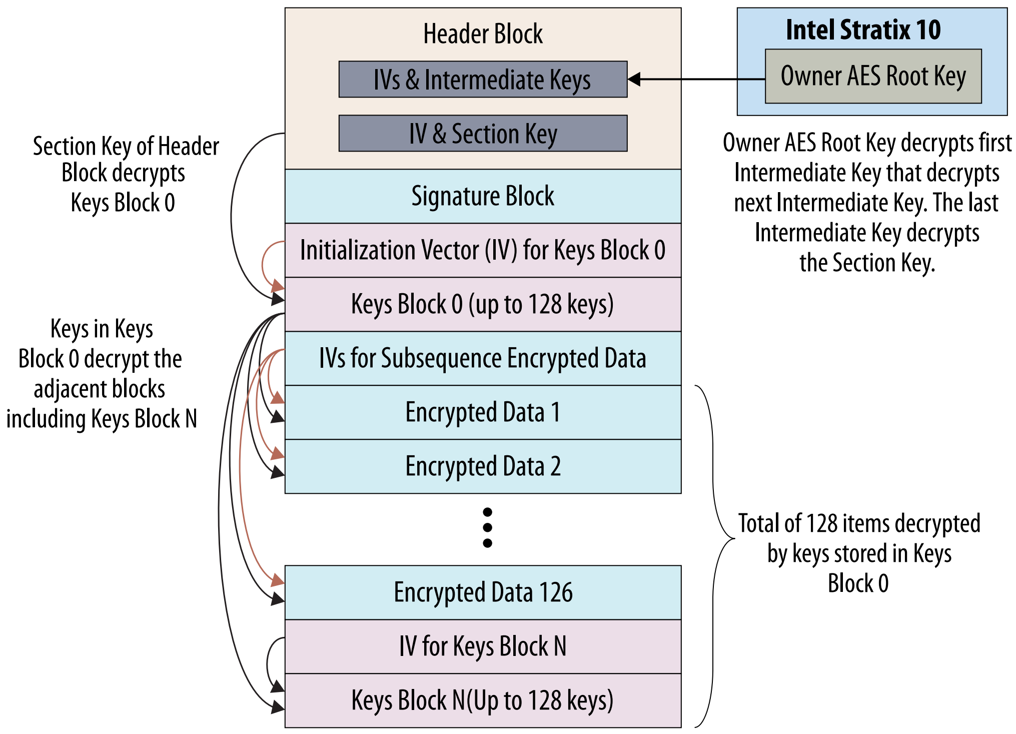
Figure 15. AES Update Mode



Decryption Process

The section key decrypts the keys block which contains up to 128 keys. Each key is 256 bits and decrypts subsequent encrypted data or another keys block.

Figure 16. Bitstream Decryption


Header Block
IVs & Intermediate Keys
IV & Section Key
Intel Stratix 10
Owner AES Root Key
Owner AES Root Key decrypts first Intermediate Key that decrypts next Intermediate Key. The last Intermediate Key decrypts the Section Key.
Section Key of Header Block decrypts Keys Block 0
Signature Block
Initialization Vector (IV) for Keys Block 0
Keys Block 0 (up to 128 keys)
Keys in Keys Block 0 decrypt the adjacent blocks including Keys Block N
IVs for Subsequence Encrypted Data
Encrypted Data 1
Encrypted Data 2
Encrypted Data 126
IV for Keys Block N
Keys Block N(Up to 128 keys)
Total of 128 items decrypted by keys stored in Keys Block 0

The initialization vector (IV) is unencrypted data that is an input to the decryption function.

Understanding Partially Encrypted Configuration Bitstreams

A partially encrypted configuration bitstream has no intermediate keys. The section key is in plaintext. The encryption finalization step creates intermediate keys and replaces the plaintext section key with an encrypted version.

Enable Scrambling

The **Enable Scrambling** parameter helps to limit any potential side-channel exposure of decrypted configuration data during the configuration process. Enabling this option places a command in the configuration bitstream that the SDM processes at configuration time. The **Enable Scrambling** parameter does not affect the encryption or decryption process.

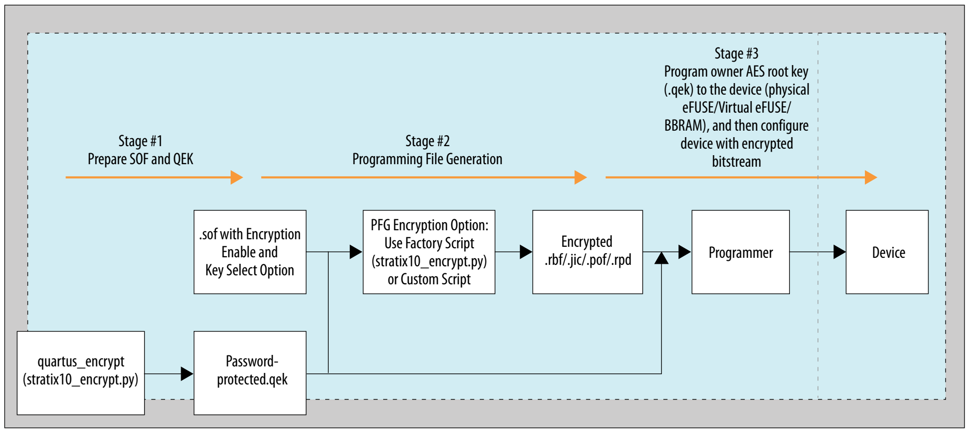
7.1. Using the Encryption Feature

Encrypting the owner image includes the following three steps:

- Step 1: Preparing the owner image and AES key files
- Step 2: Generating the programming files
- Step 3: Programming the AES key and configuring the encrypted owner image

The following flow diagram shows the processes required for each step.

Figure 17. Design Flow for Owner Image Encryption in Intel Stratix 10 Devices


Stage #1
Prepare SOF and QEK
Stage #2
Programming File Generation
Stage #3
Program owner AES root key (.qek) to the device (physical eFUSE/Virtual eFUSE/ BBRAM), and then configure device with encrypted bitstream
.sof with Encryption Enable and Key Select Option
PFG Encryption Option: Use Factory Script (stratix10_encrypt.py) or Custom Script
Encrypted .rbf/.jic/.pof/.rpd
Programmer
Device
quartus_encrypt (stratix10_encrypt.py)
Password-protected.qek

7.1.1. Step 1: Preparing the Owner Image and AES Key File

Before you generate the owner image and AES key file, you must specify authentication settings on the **Security** page of the **Device and Pin Options**.

1. On the **Security** page (**Assignments ➤ Device ➤ Device and Pin Options ➤ Security**), for **Quartus Key File** specify your root key file or signature key chain, which has the `.qky` file type.
2. Turn on the **Enable Programming Bitstream Encryption** option.
3. Specify the key storage location from **Encryption Key Select** drop-down list.
4. Generate the AES key using the `quartus_encrypt` command:

```
# This example of the quartus_encrypt command specfies the optional
# --aes_key parameter and provides the 8-word key value rather than
# allowing the quartus_encrypt command to derive the aes_key from
# the base_key

quartus_encrypt --family=stratix10 --operation=make_aes_key \
--aes_key=mykey.txt --ik_count=4 --max_key_use=32  <output.qek>
```

   As an example, for the `quartus_encrypt` command shown above, the `mykey.txt` file contains the following 8 words:

```
0xD6971FC7 0x28932CB0 0x5097E5A7 0x16968C52 0x7BB0AE8E 0x5C2F59E6
0x35B69453 0xC8E357BA
```

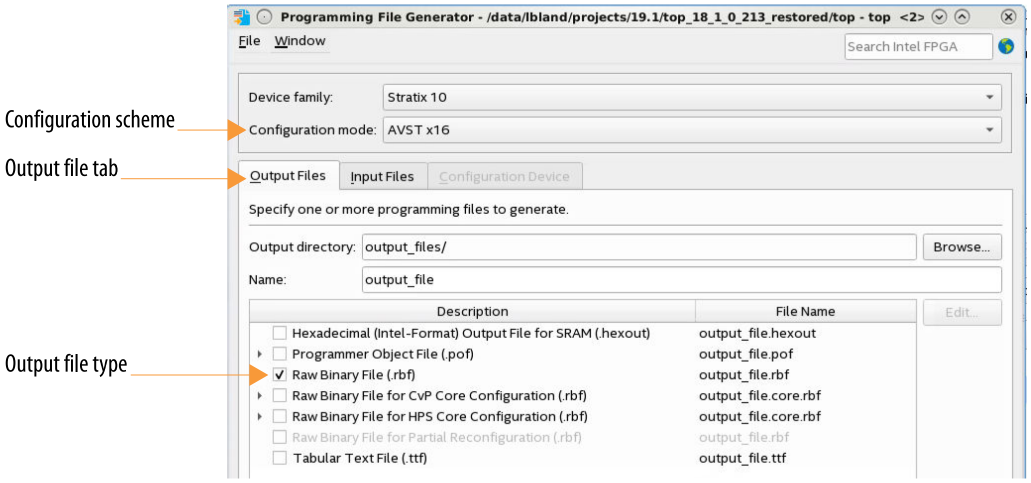
7.1.2. Step 2a: Generating Programming Files Using the Programming File Generator

You can use the Programming File Generator to encrypt and sign the owner image. The Programming File Generator supports the following signed and encrypted output file types:

- Raw Binary File (`.rbf`)
- JTAG Indirect Configuration File (`.jic`)
- Programmer Object File (`.pof`)
- Raw Programming Data File (`.rpd`)

1. On the Intel Quartus Prime File menu select **Programming File Generator**.
2. On the **Output Files** tab, specify the output file type for your configuration scheme.

Figure 18. Output File Specification



Device family: Stratix 10
Configuration mode: AVST x16

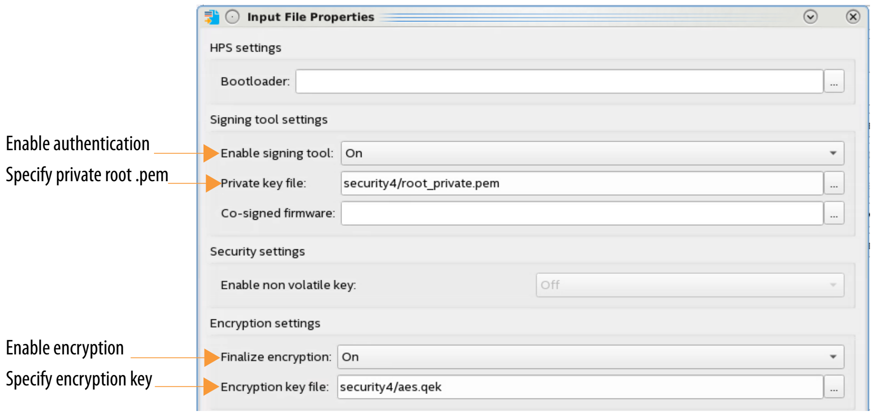
Specify one or more programming files to generate.

Output directory: output_files/
Name: output_file

	Description	File Name
☐	Hexadecimal (Intel-Format) Output File for SRAM (.hexout)	output_file.hexout
☐	Programmer Object File (.pof)	output_file.pof
✓	Raw Binary File (.rbf)	output_file.rbf
☐	Raw Binary File for CvP Core Configuration (.rbf)	output_file.core.rbf
☐	Raw Binary File for HPS Core Configuration (.rbf)	output_file.core.rbf
☐	Raw Binary File for Partial Reconfiguration (.rbf)	output_file.rbf
☐	Tabular Text File (.ttf)	output_file.ttf

3. On the **Input Files** tab, click **Add Bitstream** and browse to your `.sof`.
4. To specify encryption and authentication options select the `.sof` and click **Properties**.
   a. Turn **Enable signing tool** on.
   b. For **Private key file** select your signing key private `.pem` file.
   c. Turn **Finalize encryption** on.
   d. For **Encryption key file**, select your AES `.qek` file.

Figure 19. Input (.sof) File Properties for Authentication and Encryption



HPS settings
Bootloader:
Signing tool settings
Enable signing tool: On
Private key file: security4/root_private.pem
Co-signed firmware:
Security settings
Enable non volatile key: Off
Encryption settings
Finalize encryption: On
Encryption key file: security4/aes.qek

5. To generate the signed and encrypted bitstream, on the **Input Files** tab, click **Generate**.
   The password dialog box prompts you to input your passphrase for the `.qek`. The programming file generator generates `top.rbf` if the passphrase is correct.

7.1.3. Step 2b: Generating Programming Files Using the Command Line Interface

For JTAG or Avalon-ST configuration schemes, you can use the `quartus_pfg` script to generate the signed and encrypted output file.

1. In your output files directory, run the following command:

```
quartus_pfg -c encryption_enabled.sof top.rbf -o finalize_encryption=ON \
-o qek_file=aes.qek -o signing=ON -o pem_file=design0_sign_private.pem
```

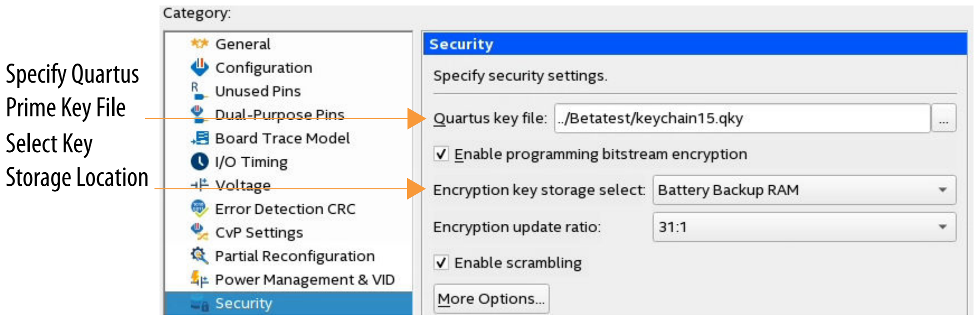
   The password dialog box prompts you to input your passphrase for the `aes.qek`. The programming file generator generates `top.rbf` if the passphrase is correct.

7.1.4. Step 3a: Specifying Keys and Configuring the Encrypted Image Using the Intel Quartus Prime Programmer

You should already have specified a storage location for your `.qek` on the **Security** page of the **Assignments ➤ Device ➤ Device and Pin Options**. In the current release, you can select **Battery Backup RAM (BBRAM)** or **eFuses**. After you make this selection, the Intel Quartus Prime Pro Edition Software identifies the `.sof` file as encryption enabled and records your settings for the **Encryption key select** and **Encryption update ratio**.

Note: If you intend to program your AES root key into physical eFuses, you must first follow the procedure in section *Storing the AES Root Key in Physical eFuses*.

Figure 20. Specify Storage Location for Encryption Key



Security

Specify security settings.

Quartus key file: ../Betatest/keychain15.qky

✓ Enable programming bitstream encryption

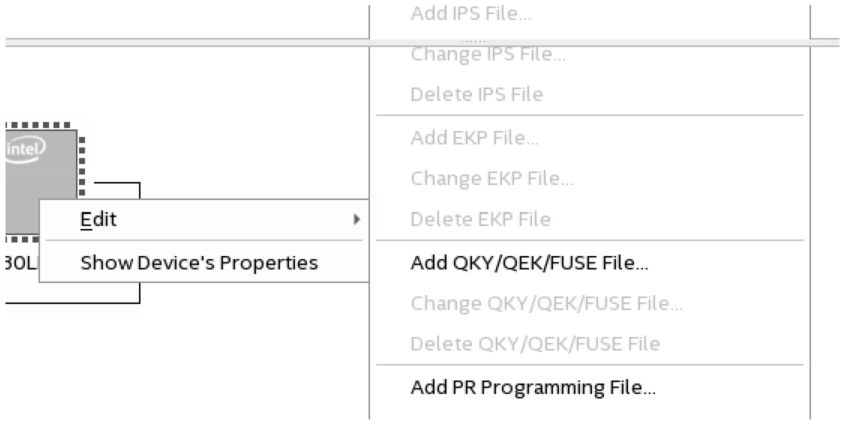
Encryption key storage select: Battery Backup RAM

Encryption update ratio: 31:1

✓ Enable scrambling

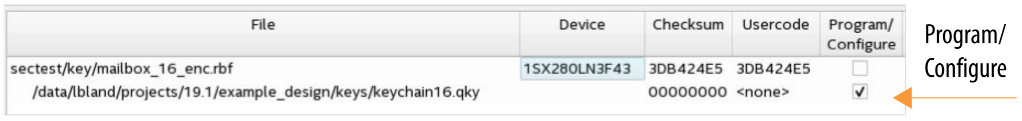
1. Bring up the Intel Quartus Prime Programmer.
2. Right click the Intel Stratix 10 device and select **Add QKY/QEK/FUSE File** file. Navigate to your `.qky` file and select it.

Figure 21. Adding .qky, .qek or .fuse files



3. Enable the **Program/Configure** option for the `.qky` file. Disable the **Program/ Configure** for any other files that may be selected. Click **Start** to program the authentication key into your Intel Stratix 10 device.

Figure 22. Program/Configure A Key File



File	Device	Checksum	Usercode	Program/ Configure
sectest/key/mailbox_16_enc.rbf	1SX280LN3F43	3DB424E5	3DB424E5	☐
/data/lbland/projects/19.1/example_design/keys/keychain16.qky		00000000	<none>	✓

Send Feedback

4. Right click the Intel Stratix 10 device and select **Add QKY/QEK/FUSE File**. Navigate to your `.qek` file and select it.
5. Enable the **Program/Configure** option for the `.qek` file. Disable the **Program/Configure** for any other files that may be selected. Click **Start**. The **Passphrase** dialog box appears. Enter your passphrase. The encryption key programs into the BBRAM, virtual eFuses or physical eFuses on the Intel Stratix 10 device.
6. With the keys programmed, you can load the signed and encrypted `.rbf` bitstream image.

Option	Description
Using the Intel Quartus Prime Programmer:	Enable the **Program/Configure** option for the `.rbf` file. Disable the **Program/Configure** for any other files that may be selected. Click **Start**.
Using a Intel MAX® 10 device or other external host:	Instruct the configuration hardware to configure the Intel Stratix 10 device from the flash memory.

If you previously programmed the authentication key into physical eFuses, it is important to remove this directive until you intend to do additional physical eFuse programming. Select **Tools ➤ Options ➤ Programmer** to restore the **Enable device security using a volatile security key** setting. Having volatile security selected ensures that you do not program physical eFuses unintentionally.

Related Information

Storing the AES Key AES in Physical eFuses on page 45

7.1.5. Step 3b: Programming the AES Key and Configuring the Encrypted Image Using the Command Line

You use the Intel Quartus Prime Programmer to program the owner AES key into the device. Then, configure the device using the encrypted bitstream.

You should already have specified a storage location for your `.qek` as explained in *Step 3a: Using the Intel Quartus Prime Programmer to Specify Keys and Configure the Device*. You can also specify this parameter using the `--key_storage argument` to the `quartus_pgm` command.

Note: If you intend to program your AES root key into physical eFuses, you must first follow the procedure in section *Storing the AES Root Key in Physical eFuses*.

1. You can program the key file using the `quartus_pgm` command:

   Caution: Incorrect programming of security eFuses can permanently prevent the device from configuring. Intel strongly recommends before programming any security eFuse that you test using the virtual eFuses to ensure that the values being programmed are correct.

   ```
   // For BBRAM
   quartus_pgm -c 1 -m jtag -o "pi;aes.qek" --key_storage "BBRAM"

   // For virtual eFuses
    quartus_pgm -c 1 -m jtag -o "pi;aes.qek" --key_storage "Virtual eFuses"

   // For physical eFuses
    quartus_pgm -c 1 -m jtag -o "pi;aes.qek" --key_storage "Real eFuses"
   ```

   The command arguments specify the following information:

   - `-c`: cable number
   - `-m`: mode
   - `-o`: operation. The argument to operation is enclosed in quotes. The letters specify the following operations:
     - `p`: program
     - `i`: load a helper image which loads the SDM firmware so that it can program the `aes.qek`
     - `;`: the argument following the *;* specifies the programming file
   - `--key_storage`: specifies the location for the encryption key. The following values are available: `BBRAM, Virtual eFuses, physical eFuses`.

2. Now program the signed encrypted bitstream using the following commands:

   ```
   quartus_pgm -c 1 -m jtag -o "p;encrypted.rbf"
   ```

Related Information

Storing the AES Key AES in Physical eFuses on page 45

7.1.6. Storing the AES Key AES in Physical eFuses

Beginning in version 19.3, the Intel Quartus Prime Pro Edition Software supports storing the AES root key in physical eFuses. In order to help protect the AES root key from extraction via physical examination of the fuses, the SDM firmware wraps the AES root key and stores the wrapped value in eFuses. You must upgrade to version 19.3 and cancel all prior Intel Firmware IDs in order to store your AES root key in physical eFuses.

After upgrading to 19.3, complete the following tasks:

1. Power on your design in version 19.3 of the Intel Quartus Prime Pro Edition Software.
2. Program Intel cancellation IDs 0-4 by programming the corresponding fuses.
3. Power cycle your system.
4. Program the AES key eFuses.
   The Intel Stratix 10 wraps the AES key.

7.1.7. Storing the AES Key in BBRAM using the JTAG Mailbox

You can use the JTAG Mailbox `VOLATILE_AES_WRITE` and `VOLATILE_AES_ERASE` commands to write the AES Key to BBRAM and erase the AES key from BBRAM.

For more information about using these commands, refer to the How can I write or erase the Intel Stratix 10 AES BBRAM encryption key using the Mailbox Client Intel FPGA IP interface and System Console?.

8. Encryption Command Detailed Description

The encryption command supports the following functions:

- Making an AES key
- Encrypting the configuration bitstream

Table 5. Signing Command Argument Summary

Argument	Options	Description
`operation`	`make_aes_key`	Generates an .qek file. Takes 6 optional arguments.
	`encrypt`	Completes the encryption process. When you enable encryption, the Bitstream Assembler always generates a partially-encrypted owner configuration bitstream (`.rbf`) from the `.sof` input file.

8.1. Make AES Key

You can use the six optional arguments to the `quartus_encrypt` command to customize encryption.

Command	`quartus_encrypt --family=stratix10 --operation=make_aes_key <output .qek>`
Input file	None
Output file	`.qek`
Arguments	This command includes the following 6 optionals arguments: • `ik_count`: Specifies the number of intermediate keys to encrypt the owner configuration bitstream. The default value is 3. • `max_key_use`: Specifies the maximum number of keys to use to encrypt the owner configuration bitstream. The default value is 128. The following restriction applies to the total number of encryption keys: $\log_2(\text{max\_key\_use}) * (\text{ik\_count} + 1) < 64$ • `passphrase`: Specifies an optional file path that contains a passphrase to protect the `.qek`. If you do not specify this argument, the `quartus_encrypt` command prompts you to enter the `passphrase`. • `base_key`: Specifies a binary file as the `base_key` to generate a root key if you do not specify an `aes_key` or key derivation key. If you do not specify a `base_key`, `quartus_encrypt` uses random data. • `aes_key`: Specifies an AES key in hexadecimal words. If you do not specify an `aes_key`, the `quartus_encrypt` command derives the `aes_key` from the `base_key`.

8.2. Encrypt the Bitstream

You enable encryption on the **Security** page of the **Assignments ➤ Device ➤ Device and Pin Options ➤ Security** tab. When you enable encryption, the Bitstream Assembler always generates a partially-encrypted owner configuration bitstream. A partially encrypted configuration bitstream has no intermediate keys. The section key is in plaintext. The encryption finalization step creates intermediate keys and replaces the plaintext section key with an encrypted version. The `quartus_encrypt` command completes the encryption process.

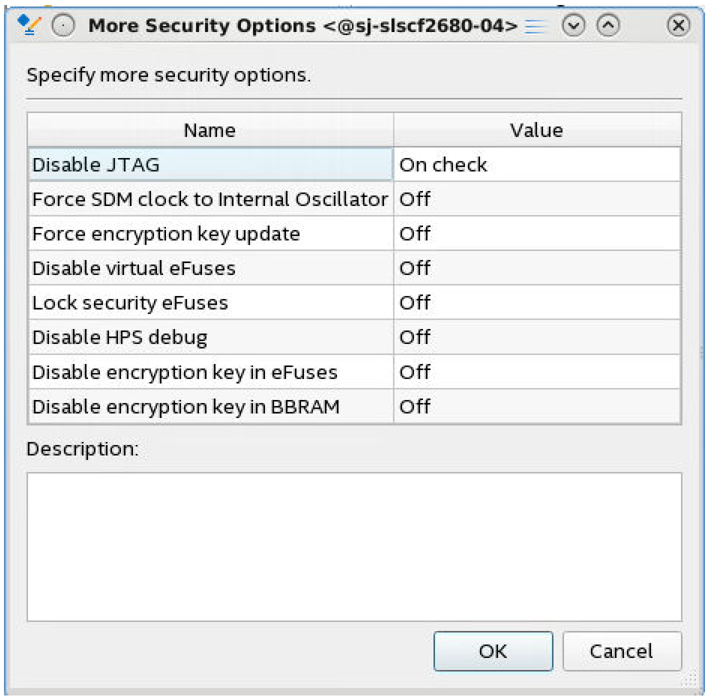
Command	`quartus_encrypt --family=stratix10 --operation=encrypt --Key=<output .qek> \ <partially-encrypted .rbf> <fully encrypted .rbf>`
Input file	Partially encrypted `.rbf` file.
Output file	Fully encrypted `.rbf` file.
Arguments	This command includes the following 2 required arguments: • `key`: Specifies a `.qek` file. • `passphrase`: Specifies an optional file path that a contains passphrase to protect the `.qek`. If you do not specify this argument, the `quartus_encrypt` command prompts you to enter the `passphrase`.

9. Using eFuses

Intel Quartus Prime Pro Edition devices use eFuses to permanently store device and security information. Owner eFuse fields are quadruple-redundant. The Intel Quartus Prime Pro Edition Programmer programs eFuses using a JTAG interface. eFuse programming is an irreversible process in which a large amount of electrical current passes through a small chip feature until the feature is destroyed.

You can set many of the eFuses using the **More Options** button of the **Security** category on the **Assignments ➤ Device ➤ Device and Pin Options** menu as shown.

Figure 23. Setting Security Options Using eFuses



Specify more security options.

Name	Value
Disable JTAG	On check
Force SDM clock to Internal Oscillator	Off
Force encryption key update	Off
Disable virtual eFuses	Off
Lock security eFuses	Off
Disable HPS debug	Off
Disable encryption key in eFuses	Off
Disable encryption key in BBRAM	Off

Description:

The following table describes all available owner eFuses.

Table 6. Owner Programmable eFuses

eFuse Name	Legal Values	Description
Intel FPGA public key hash	384-bit hex	Read-only.
Intel FPGA public key cancellation	32-bit hex	32 Intel cancellation IDs are available. Each bit corresponds to the cancellation ID.

continued...

eFuse Name	Legal Values	Description
Co-signed firmware	1-bit boolean	When you program this fuse, both you and Intel must sign the device firmware. Intel signs the device firmware with the root public key during the manufacturing process.
Device not secure	1-bit boolean	If you receive a device and this fuse is programmed do not use the device and contact Intel.
Owner encryption key program done	1-bit boolean	The Programmer programs the owner AES key into eFuses.
Owner encryption key program start	1-bit boolean	
Owner key cancellation	32-bit hex	0-31. The Intel Stratix 10 device has 4 redundant cancellation bits of with each fuse. The Programmer programs all 4 copies when you cancel the corresponding fuse.
Owner root public key hash	384-bit hex	Stores the hash value of the owner root public key.
Owner public key size	[0, 256, 384]	Specifies owner public key size. Intel recommends using 384-bit keys if possible.
JTAG disable	1-bit boolean	When set, disables JTAG command and configuration. Setting this eFuse eliminates JTAG as mode of attack, but also eliminates boundary scan.
Force SDM clock to **Internal Oscillator**	1-bit boolean	When set, disables an external clock source for the SDM for bitstream configuration. Forcing the SDM to use an internal oscillator helps to limit potential interruptions or attacks by modifying an external clock during configuration.
Force encryption key update	1-bit boolean	When set, all encrypted bitstreams must specify the **Encryption Update Ratio** on the Intel Quartus Prime **Assignments ➤ Device and Pin Options ➤ Security** menu.
Disable virtual eFuses	1-bit boolean	When set, disables the eFuse virtual programming capability.
Lock security eFuses	1-bit boolean	Programming this fuse prevents the future programming of any owner-accessible security policy fuses, not including key cancellation ID fuses.
Disable HPS debug	1-bit boolean	When set, permanently disables debugging using JTAG to access the HPS.
eFuse key disable	1-bit boolean	When set, the device does not use an AES key stored in eFuses.
BBRAM key disable	1-bit boolean	When set, the device does not use AES key stored in BBRAM.

Simulating the Public Key Hash Virtual eFuses

Because eFuses are non-volatile, Intel recommends validating eFuse programming before programming physical eFuses on the Intel Stratix 10 device.

This example provides the steps to validate the public key hash. First, you program the public key hash value into in virtual eFuses. Then, you compare that value to the value that the **Examine** function stores in `hash_fuse.txt`:

1. Turn on **Enable device security using a volatile security key** in the Intel Quartus Prime Programmer. When you select this option the Intel Quartus Prime Pro Edition stores the eFuse values in firmware registers.
2. In the Intel Quartus Prime Programmer click **Add File** and browse to your signed bitstream.
3. In the Intel Quartus Prime Programmer turn on the **Program/Configure** and **Examine** options.
4. Click **Start**.
5. After programming completes, the Programmer displays the hash value of the signature key stored in firmware. You can now compare that value to the value you generate by creating a `hash_fuse.txt` file using the `quartus_sign` command with the operation set to `fuse_info`.

Related Information

Step 5b: Calculating the Owner Root Public Key Hash on page 26

9.1. Fuse Programming Input Files

The Intel Quartus Prime Programmer supports the following three input file types for fuse programming. Intel Quartus Prime key file (`.qky`), the Intel Quartus Prime encryption key (`.qek`), and the `.fuse` file.

The files provide the following information to the Intel Quartus Prime Programmer:

- `.qky`: Provides the owner public root key for authentication and the second-level key for firmware authentication. Use this file for the following functions:
  - To program and verify the public root key fuses
  - To sign the owner configuration bitstream
  - To sign the device firmware
  - To sign the HPS debug certificate
- `.qek`: Provides the AES key for encryption. Use this file for the following functions:
  - To program the AES key fuses
  - To encrypt the owner configuration bitstream
- `.fuse`: Specifies all owner fuses. Also includes the public root key which is read-only. Use this file for the following functions:
  - To program and verify security fuses
  - To program owner-defined fuses

9.1.1. Fuse File Format

The `.fuse` file contains a list of fuse name-value pairs. The value specifies whether the fuse has been programmed (blown) and its cancellation ID.

The following example shows the format of the `.fuse` file.

```
# Comment
<fuse name> = <value>
<fuse name> = <value>
<fuse name> = <value>
```

You can use the Intel Quartus Prime Programmer **Examine** option to read all currently programmed fuses in the Intel Stratix 10 device and store this information in a `.fuse` file.

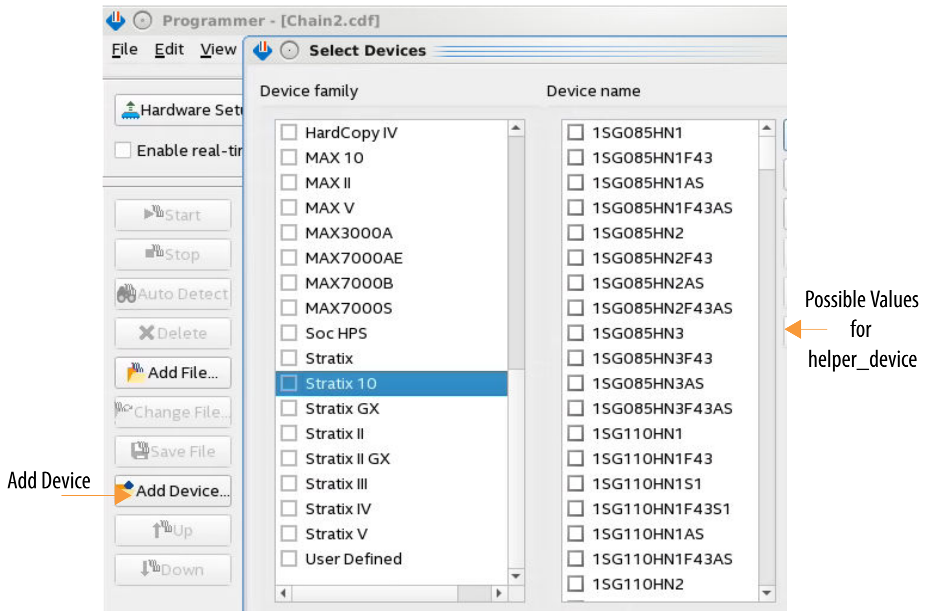
9.1.2. Programming eFuses

You can program eFuses to enable or disable device security features. Before programming eFuses you must check the current state of eFuse programming for your device. This procedure ensures that you add the new eFuse commands to the existing eFuse programming commands, if any.

The example commands specify the `helper_device 1SX280LH2`. If you are using a different Intel Stratix 10 device, provide the appropriate ordering code for that device up to the speed grade designation. Helper images are necessary for flash and fuse programming using the Intel Quartus Prime Programmer. The helper image programs the SDM firmware.

1. To find the list of helper devices, in the Intel Quartus Prime Programmer, select **Add Device.**
2. In the **Device family** list, select **Intel Stratix 10**. In the **Device name** list, identify the find the part number that matches your device.

Figure 24. User the Programmer to Determine the helper_device Argument



3. Generate an unsigned helper image for eFuse programming.

```
quartus_pfg --helper_image -o helper_device=1SG280HN2 -o subtype=FUSE \
 -o fw_source=nadder.zip unsigned_helper_image.rbf
```

4. Configure your Intel Stratix 10 device with the `helper_image.rbf` file you just created.

```
quartus_pgm -c 1 -m jtag -o “p;unsigned_helper_image.rbf”
```

5. Generate the current device fuse status file, `programming_file.fuse`.

```
quartus_pgm -c 1 -m jtag -o “e;programming_file.fuse;1SX280LH2”
```

6. Edit `programming_file.fuse` to add the required eFuses. There are four copies of eFuses. Programming changes all 4 copies from 0 to 1. Add the following command to `programming_file.fuse`.

```
<fuse_name> = "0xF"
```

7. Program the eFuses:

```
//For physical (non-volatile) eFuses
quartus_pgm -c 1 -m jtag -o "p;programming_file.fuse" --non_volatile_key
```

```
//For virtual (volatile) eFuses
quartus_pgm -c 1 -m jtag -o "p;programming_file.fuse"
```

Related Information

Intel Stratix 10 GX/SX Device Overview
For an explanation of Intel Stratix 10 device ordering codes.

9.1.3. Canceling eFuses

If you have already programmed the owner root key hash into eFuses, you must manually cancel IDs 0, 2, and 3 in the FPGA. When you programmed the owner root key hash into eFuses, ID 1 was automatically canceled.

Follow these steps to cancel eFuses that specify SDM firmware versions that are no longer valid.

1. Extract the existing fuse information by running the following command-line command:

   ```
   quartus_pgm -c 1 -m jtag -o "ie;my_fuse.fuse;1SX280LH2"
   ```

   This command generates a `my_fuse.fuse` text file.

   Sample contents of `my_fuse.fuse`:

   ```
   # Co-signed firmware                  = "0xF"
   # Device not secure                   = "0x0"
   # Intel key cancellation              = ""
   # Owner fuses                         =
   "0x00000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000
   00000000000000000000000000000000000000000"
   # Owner key cancellation              = ""
   # Owner public key hash               =
   "0x00000000000000000000000000000000000CE520B15B082E67ACEBCB8545CE239FDBB8CDE60
   83F6DF9D3BF542932EA5039"
   # Owner public key size               = "0xF"
   # QSPI start up delay                 = "0x0"
   # SDMIO0 is I2C                       = "0x0"
   ```

2. Using a text editor, update `my_fuse.fuse` to specify the keys to cancel. Change:

   ```
   #Intel key cancellation               = ""
   ```

   to:

   ```
   Intel key cancellation               = "0,1,2,3,4"
   ```

   Note: Be sure to remove the initial #.

3. Run the following command to program the cancellation ID eFuses:

   ```
   //For physical (non-volatile) eFuses
   quartus_pgm -c 1 -m jtag -o "p;my_fuse.fuse" --non_volatile_key
   ```

   ```
   //For virtual (volatile) eFuses
   quartus_pgm -c 1 -m jtag -o "p;my_fuse.fuse"
   ```

9.1.4. Converting Key, Encryption, and Fuse Files to Jam Staple File Formats

You can use the `quartus_pfg` command-line command to convert `.qky`, `.qek`, and `.fuse` files to Jam Standard Test and Programming Language (STAPL) Format File (`.jam`) and Jam Byte Code File (`.jbc`). You can use these files to program Intel FPGAs using the Jam STAPL Player and the Jam STAPL Byte-Code Player, respectively.

A single `.jam` or `.jbc` contains several functions including a firmware helper image and program, blank check, and verification of key and fuse programming.

Caution: When you convert the AES `.qek` file to `.jam` format, the `.jam` file contains the AES key in plaintext but obfuscated form. Consequently, you must protect the `.jam` file when storing the AES key. You can protect the `.jam` file by provisioning the AES key in a secure environment.

quartus_pfg Conversion Commands

Here are examples of `quartus_pfg` conversion commands:

```
quartus_pfg -c -o helper_device=1SX280LH2 root.qky RootKey.jam
quartus_pfg -c -o helper_device=1SX280LH2 root.qky RootKey.jbc
quartus_pfg -c -o helper_device=1SX280LH2 nd.qek nd_qek.jam
quartus_pfg -c -o helper_device=1SX280LH2 nd.qek nd_qek.jbc
quartus_pfg -c -o helper_device=1SX280LH2 cancel_id.fuse nd_fuse.jam
quartus_pfg -c -o helper_device=1SX280LH2 cancel_id.fuse nd_fuse.jbc
```

For more information about the using the Jam STAPL Player for device programming refer to *AN 425: Using the Command-Line Jam STAPL Solution for Device Programming*.

Using the .jam Files to Program Root Key and AES Encryption Key

The run the following commands to program the owner root public key and AES encryption key:

```
// To load the helper bitstream into the FPGA.
// The helper bitstream include SDM firmware
quartus_jli -c 1 -a CONFIGURE RootKey.jam

// To program the owner root public key into virtual eFuses
auartus_jli -c 1 -a PUBKEY_PROGRAM RootKey.jam

//To program the owner root public key into physical eFuses
quartus_jli -c 1 -a PUBKEY_PROGRAM -e DO_UNI_ACT_DO_EFUSES_FLAG RootKey.jam

// To program the AES Encryption key into BBRAM
quartus_jli -c 1 -a AESKEY_PROGRAM -e DO_UNI_ACT_DO_BBRAM_FLAG EncKey.jam
```

Related Information

AN 425: Using the Command-Line Jam STAPL Solution for Device Programming

10. Document Revision History for Intel Stratix 10 Device Security User Guide

Document Version	Intel Quartus Prime Version	Changes

2020.01.15	19.3	Corrected the `pem_file` argument in *7.1.3. Step 2b: Generating Programming Files Using the Command Line*. The correct command uses `pem_file=design0_sign_private.pem`:
		`quartus_pfg -c encryption_enabled.sof top.rbf \`
		`-o finalize_encryption=ON -o qek_file=aes.qek \`
		` -o signing=ON -o pem_file=design0_sign_private.pem`

2020.01.06	19.3	Made the following changes: • Corrected the `quartus_encrypt` command in the *Step 1: Preparing the Owner Image and AES Key File*topic. The `ik_count` and `max_key_use` arguments must be preceded by `--`. • Added command showing how to convert an `.rbf` to `.jam` format in the *Step 4: Signing the Bitstream* topic. • Added the following note to the *Converting Key, Encryption, and Fuse Files to Jam Staple File Formats* topic: ***Caution:*** When you convert the AES `.qek` file to `.jam` format, the `.jam` file contains the AES key in plaintext but obfuscated form. Consequently, you must protect the `.jam` file when storing the AES key. You can protect the `.jam` file by provisioning the AES key in a secure environment. • Added a link to the How can I write or erase the Intel Stratix 10 AES BBRAM encryption key using the Mailbox Client Intel FPGA IP interface and System Console? article in *Storing the AES Key in BBRAM using the JTAG Mailbox*.
2019.10.30	19.3	Added the following new security features: • Added support for physical (non-volatile) eFuses. • Changed the way you specify virtual (volatile) or physical (non-volatile) eFuses. The `--non_volatile_key` parameter is now an argument to the `quartus_pgm` command. Consequently, you no longer need to recompile to change the eFuse storage location. • Increased the number of public keys entries supported from 2 to 3. • Added support for a signed secure HPS debug certificate to prevent unauthorized remote or physical access to the HPS. • Decreased the encryption update ratio from 127:1 to 31:1. • Revised description the *Using the Authentication Feature* example. The example now specifies permission 6 to allow the key to sign both the Core (permission=2) and HPS (permission=4) sections of the configuration bitstream. You must create separate key chains to limit the permissions to either Core or HPS. • Added support for 10 additional eFuses described in the *Owner Programmable eFuses* table. • Added examples of advanced security features. • Added descriptions of side-channel mitigation features.

continued...

Document Version	Intel Quartus Prime Version	Changes
		• Added the following topics: — *Step 4a: Protecting the AES Key when Storing the AES in eFuses* — *Step 4b: Protecting the AES Key when Storing the AES Key in BBRAM* — *Encryption Command Detailed Description* — *Make AES Key* — *Encrypt the Bitstream* — *Programming eFuses* — *Canceling eFuses* • Added examples of `.jam` commands under the *Using the .jam Files to Program Root Key and AES Encryption Key* heading. • Corrected *AES Update Mode* figure. The number of data bits in a data block is 256, not 128. • Corrected the cancellation ID Numbers in *Figure 5: Three-Key Signature Chain*. The cancellation IDs are 0 and 1. • Removed recommendation to use separate signing keys for core and HPS in Intel Stratix 10 SX devices. Changed *Using the Authentication Feature* example to set permissions to 6 which can sign both the core and HPS. • Revised *Anti-Tampering* topic. • Revised the*Using eFuses* topic. • Corrected minor errors and typos.
2019.05.30	19.1	Made the following corrections: • Corrected the *Signing Command Argument Summary* table. The references to `.key` format should say `.qky` format.
2019.05.10	19.1	Made the following corrections: • Removed spaces before the fuse programming file name in the `quartus_pgm` commands in *Step 3b: Programming the AES Key and Configuring the Encrypted Image Using the Command Line*. • Changed file name argument to `-o "p;my_fuse.fuse"` in *Step 4* of *Canceling Non-Volatile eFuses*.
2019.05.07	19.1	Initial release.