Contents

1. Important Notice to Customers Regarding Features Added in Intel® Quartus® Prime Pro Edition Software Version 20.1 .............................................................. 4

2. Intel Stratix® 10 Device Security Overview ...................................................................................... 5
   2.1. Intel Stratix 10 Secure Device Manager (SDM) .............................................................. 7
   2.2. Enabling Intel Stratix 10 Security Features ................................................................. 9
   2.3. Owner Security Keys and Storage Options ................................................................... 9
       2.3.1. Owner AES Key Programming .............................................................................. 11
   2.4. Planned Security Features ......................................................................................... 11
       2.4.1. Black Key Provisioning ...................................................................................... 11

3. Design Authentication and Signature Chains .............................................................................. 13
   3.1. Design Authentication Overview ............................................................................... 13
       3.1.1. The Configuration Bitstream .............................................................................. 13
       3.1.2. Signature Block ................................................................................................. 15
   3.2. Creating a Signature Chain ....................................................................................... 19
       3.2.1. Step 1: Creating the Root Key ............................................................................. 21
       3.2.2. Step 2: Creating the Design Signing Key ............................................................ 21
       3.2.3. Step 3: Appending the Design Signature Key to the Signature Chain ................. 22
       3.2.4. Step 4: Signing the Bitstream ............................................................................. 23
       3.2.5. Step 4a: Signing the Bitstream Using the Programming File Generator ............. 23
       3.2.6. Step 4b: Signing the Bitstream Using the quartus_sign Command ...................... 26
       3.2.7. Step 5: Programming the Owner Root Public Key for Authentication ................. 27
       3.2.8. Step 5a: Programming the Owner Root Public Key .............................................. 27
       3.2.9. Step 5b: Calculating the Owner Root Public Key Hash ....................................... 29
       3.2.10. Verifying a Configuration Bitstream Signature .................................................. 29
   3.3. Co-Signing Device Firmware Overview ........................................................................... 32
       3.3.1. Using the Co-Signing Feature ............................................................................. 32
   3.4. Signing Command Detailed Description ....................................................................... 34
       3.4.1. Generate Private PEM Key .................................................................................. 35
       3.4.2. Generate Public PEM Key .................................................................................. 35
       3.4.3. Generate Root Signature Chain .......................................................................... 36
       3.4.4. Append Key to Signature Chain .......................................................................... 36
       3.4.5. Sign the Bitstream, Firmware, or Debug Certificate ........................................... 37
       3.4.6. Calculate Root Public Key Hash from QKY .......................................................... 37

4. Encryption and Decryption ............................................................................................................. 38
   4.1. Using the Encryption Feature ....................................................................................... 40
       4.1.1. Step 2a: Generating Programming Files Using the Programming File Generator 40
       4.1.2. Step 2b: Generating Programming Files Using the Command Line Interface .... 42
       4.1.3. Step 3a: Specifying Keys and Configuring the Encrypted Image Using the Intel Quartus Prime Programmer ............................................................ 42
       4.1.4. Step 3b: Programming the AES Key and Configuring the Encrypted Image Using the Command Line ................................................................. 44
       4.1.5. Storing the AES Key in Physical eFuses ............................................................... 45
       4.1.6. Storing the AES Key in BRAM using the JTAG Mailbox .................................... 45
   4.2. Using a PUF-Wrapped AES Key (Beta) ............................................................................ 45
4.2.1. Step 1: Enrolling the Intrinsic ID PUF and Wrapping the AES Key....................46
4.2.2. Step 2: Querying Intrinsic ID PUF Activation Status...................................... 47
4.2.3. Step 3a: Specifying the Output Files for Flash Programming .......................... 48
4.2.4. Step 3b: Specifying Input Files for Flash Programming..................................49
4.2.5. Step 3c: Specifying the Configuration Device for Flash Programming...........50
4.2.6. Step 4: Programming the PUF-Wrapped Key to Quad SPI Flash...................... 52
4.2.7. Enabling and Disabling the More Security Options.........................................53
4.2.8. Additional Information About Using the PUF................................................54
4.3. Encryption Command Detailed Description.............................................................. 55
4.3.1. Make AES Key..............................................................................................55
4.3.2. Encrypt the Bitstream..................................................................................56

5. Anti-Tamper Monitoring and Mitigation (Beta)..............................................................57
5.1. Specifying Anti-Tamper Pins................................................................................ 57
5.2. Specifying the Anti-Tamper Detection Parameters..............................................58
5.3. Understanding the Anti-Tamper Responses......................................................61
5.4. Understanding the Zeroization Tamper Response ... .............................................61

6. Using eFuses.................................................................................................................63
6.1. eFuses Overview ...............................................................................................63
6.1.1. Fuse Programming Input Files..................................................................65
6.1.2. Understanding Fuse File Format...............................................................66
6.1.3. Programming eFuses ................................................................................66
6.1.4. Canceling eFuses......................................................................................68
6.1.5. Converting Key, Encryption, and Fuse Files to the Jam Staple File Formats....68

7. Additional Security Features........................................................................................70
7.1. JTAG Disable....................................................................................................70
7.2. Using an HPS Debug Certificate.......................................................................70
7.3. Enabling HPS JTAG Debugging ..........................................................................71


A. Appendices..................................................................................................................77
A.1. Acronyms and Glossary....................................................................................77
A.2. File Types for Security..................................................................................78
A.3. quartus_pgm Command Operation Argument................................................79
1. Important Notice to Customers Regarding Features Added in Intel® Quartus® Prime Pro Edition Software Version 20.1

The Intrinsic ID Physically Unclonable Function (PUF)-based AES key storage and Anti-Tamper features are a beta release in Intel® Quartus® Prime Pro Edition software version 20.1. The documentation of these features should be considered preliminary and subject to change.

For more information, please contact your Intel sales representative or Intel support.
2. 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.

The following security features are available to protect Intel Stratix 10 devices and your intellectual property:

**Authentication**

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. Authentication uses following technology to protect your intellectual property and device:

- **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.

  eFuse programming to establish authentication requirements before configuration completes: You 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.
Encryption

Encryption helps to protect confidential information such as intellectual property or sensitive data from being extracted from the owner configuration bitstream. Encryption uses following technology to protect your device and intellectual property:

- Advanced Encryption Standard (AES)-256 encryption, counter mode: This feature helps protect the confidentiality of intellectual property (IP) or sensitive data in the owner configuration bitstream. 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.
- Advanced technology to store the AES root key choices: Intel Stratix 10 devices currently support the following storage locations for root AES keys:
  - Battery backup RAM (BBRAM)
  - eFuses: virtual or physical
  - Quad SPI flash memory: wrapped in an Intrinsic* ID (IID) physically unclonable function (PUF) (Beta)

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.

<table>
<thead>
<tr>
<th>Intel Stratix 10</th>
<th>Authentication</th>
<th>Advanced Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>GX</td>
<td>Yes</td>
<td>-AS suffix devices</td>
</tr>
<tr>
<td>SX</td>
<td>Yes</td>
<td>-AS suffix devices</td>
</tr>
<tr>
<td>MX</td>
<td>Yes</td>
<td>-AS suffix devices</td>
</tr>
<tr>
<td>TX</td>
<td>Yes</td>
<td>-AS suffix devices</td>
</tr>
<tr>
<td>DX</td>
<td>Yes</td>
<td>Yes, all devices</td>
</tr>
</tbody>
</table>
Additional Security Features

In addition to authentication and encryption Intel Stratix 10 devices support the following features to help secure your device and intellectual property:

- Anti-tamper protection (Beta)
- eFuse support for security settings
- Secure HPS debugging
- JTAG disablement

The Related information section provides links to the topics that describe these features.

Related Information

- JTAG Disable on page 70
- Enabling HPS JTAG Debugging on page 71
- Anti-Tamper Monitoring and Mitigation (Beta) on page 57
  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.

2.1. Intel Stratix 10 Secure Device Manager (SDM)

The Secure Device Manager (SDM) is a triple-redundant processor-based module that manages Intel Stratix 10 device configuration and security. The SDM authenticates and decrypts configuration data.
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
2.2. 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.

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**

eFuses Overview on page 63

2.3. Owner Security Keys and Storage Options

Intel Stratix 10 devices support two types of security keys: The owner root public key hash and the owner AES key.

**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, BBRAM, or in quad SPI flash as a PUF-wrapped key.

Table 1. Comparison of AES Key Storage Options

<table>
<thead>
<tr>
<th>Storage Option</th>
<th>Customer Access</th>
<th>Persistence</th>
<th>Typical Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual eFuses</td>
<td>Write/Erase</td>
<td>Volatile. Power cycling erases values.</td>
<td>Use only to validate eFuses settings. The SDM can read eFuses without powering up the physical eFuses. Virtual eFuses replicate physical eFuse behavior until you power cycle the pcb, at which time all virtual eFuses reset. Intel strongly recommends using virtual eFuses during development and physical eFuses for production designs. Do not rely on virtual eFuses to maintain keys security. Use test keys when working with virtual eFuses.</td>
</tr>
<tr>
<td>Physical eFuses</td>
<td>Write once</td>
<td>Non-volatile. System must be powered on to read values.</td>
<td>Specifies permanent security policy. Prevents potential attacks that gain access to the device by changing security settings.</td>
</tr>
<tr>
<td>Battery Backup RAM</td>
<td>Write/Erase</td>
<td>Non-volatile. Uses dedicated V_{CCBAT} power supply. The voltage range is 1.2V - 1.8V.</td>
<td>Supports re-provisioning an AES key. Can reprogram as many times as necessary. After powering down the pcb, disconnecting the battery ensures that the key is erased.</td>
</tr>
<tr>
<td>IID PUF-wrapped and stored in Quad SPI Flash (Beta)</td>
<td>Write/Erase</td>
<td>Non-volatile.</td>
<td>Stores the AES key in external flash memory, encrypted with a secure device-unique key. Supports re-provisioning the AES key without requiring battery backup.</td>
</tr>
</tbody>
</table>

Note: The BBRAM key vault includes a built-in function to perform periodic key flipping to prevent key imprinting.

You program both the root public key hash and the AES key using JTAG. The configuration bitstream records the owner AES key location. For extra security, you can program eFuses to disable the other key storage locations. For example, if your design stores the AES key in eFuses, you can program the Disable encryption key in BBRAM and Disable PUF-wrapped encryption key eFuses for additional security.

Intel Stratix 10 devices support both red key (unencrypted) and black key (encrypted) provisioning (transport). Red key provisioning transmits keys over JTAG 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 11 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.

---

(1) Once you program the AES key, you cannot change or reprogram the AES key.

(2) For more information about required voltage ranges refer to the Intel Stratix 10 Device Family Pin Connection Guidelines.
2.3.1. Owner AES Key Programming

You specify the storage option for the AES root key on the Security page of the Assignments ➤ Device ➤ Device and Pin Options. 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 2. Specify Your Encryption Key

The Intel Quartus Prime Programmer also includes an Encryption Key Select option with three choices: Battery Backup RAM, eFuses, or Quad SPI Intrinsic ID PUF-wrapped. 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.

2.4. Planned Security Features

Features mentioned in this section are planned for a future release of Intel Quartus Prime Pro Edition software.

2.4.1. 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 (also called 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. Your HSM may be located at a secure site which is connected to the Intel Stratix 10 device via the Internet. In such cases 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.
3. Design Authentication and Signature Chains

3.1. Design Authentication Overview

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 for 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 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.

In the Intel Quartus Prime Software release 20.1 and earlier releases, selecting 384-bit keys results in a SHA384 hash being used in the authentication blocks in the bitstream. However, the remainder of the bitstream uses SHA256 for integrity checking. Support for SHA384 hashes in the bitstream is planned to be supported in a future release.

3.1.1. 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 3. Example of an Intel Stratix 10 Configuration Bitstream Structure**

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.
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.

### 3.1.2. 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 5. 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 7 on page 20.

**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 2. Signature Block

<table>
<thead>
<tr>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-384 hash of header block</td>
<td>This hash function checks for accidental changes in the preceding block of the configuration bitstream, typically the header block.</td>
</tr>
<tr>
<td>Signature chains</td>
<td>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.</td>
</tr>
<tr>
<td>Dynamic sector pointers</td>
<td>Locate the design sections for the remainder of the image when you store the image in flash memory.</td>
</tr>
<tr>
<td>32-bit CRC</td>
<td>Protects the block from accidental modification. The CRC does not provide security. Software tools can check the CRC to identify accidental modifications.</td>
</tr>
</tbody>
</table>

### 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 3. Signature Chain Content

<table>
<thead>
<tr>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Key Entry</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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 17 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 authenticifies 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.
3.1.2.1. 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 load the most current firmware. This section describes when and how Intel firmware IDs are canceled.

As of Intel Quartus Prime Pro Edition Version 20.1, Intel has used the following firmware IDs.

Table 4. Intel Firmware IDs

<table>
<thead>
<tr>
<th>Firmware ID</th>
<th>Firmware Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>Early versions of firmware</td>
</tr>
<tr>
<td>4</td>
<td>Intel Quartus Prime Pro Edition 19.1 and 19.2</td>
</tr>
<tr>
<td>5</td>
<td>Intel Quartus Prime Pro Edition 19.3 and 19.4</td>
</tr>
<tr>
<td>6</td>
<td>Intel Quartus Prime Pro Edition 20.1</td>
</tr>
</tbody>
</table>

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 20.1 firmware to program the public key hash, this firmware automatically cancels IDs 0 to 5. 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.

### 3.1.2.2. 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](https://www.intel.com) 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

### 3.2. Creating a Signature Chain

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

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 7. Steps to Create a Signature Chain**
3.2.1. 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>On the Start menu, point to Programs ➤ Intel FPGA ➤ Nios II EDS ➤ &lt;version&gt; and click Nios II &lt;version&gt; Command Shell.</td>
</tr>
<tr>
<td>Linux</td>
<td>In a command shell change to the &lt;install_dir&gt;/nios2eds and run the following command:</td>
</tr>
<tr>
<td></td>
<td>./nios2_command_shell.sh</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>With passphrase</td>
<td>quartus_sign --family=stratix10 --operation=make_private_pem -- curve=&lt;prime256v1 or secp384r1&gt; &lt;root_private.pem&gt; Enter the passphrase when prompted to do so.</td>
</tr>
<tr>
<td>Without passphrase</td>
<td>quartus_sign --family=stratix10 --operation=make_private_pem -- curve=&lt;prime256v1 or secp384r1&gt; --no_passphrase &lt;root_private.pem&gt;</td>
</tr>
</tbody>
</table>

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>
```

3.2.2. 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 as the one you specified for the root key.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>With passphrase</td>
<td><code>quartus_sign --family=stratix10 --operation=make_private_pem --</code></td>
</tr>
<tr>
<td></td>
<td><code>curve=&lt;prime256v1 or secp384r1&gt; &lt;design0_sign_private.pem&gt;</code></td>
</tr>
<tr>
<td></td>
<td>Enter the passphrase when prompted to do so.</td>
</tr>
<tr>
<td>Without passphrase</td>
<td><code>quartus_sign --family=stratix10 --operation=make_private_pem --</code></td>
</tr>
<tr>
<td></td>
<td><code>curve=&lt;prime256v1 or secp384r1&gt; --no_passphrase</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;design0_sign_private.pem&gt;</code></td>
</tr>
</tbody>
</table>

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.

3.2.3. 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 22, 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.

### 3.2.4. Step 4: Signing the Bitstream

Once you generate the private PEM and .qky files, you are ready to sign the bitstream. There are several 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 (.rpdd) formats are available for Active Serial (AS) configuration. The Programmer Object File (.pof) and .rbf are available for Avalon® Streaming (Avalon-ST) configuration.

- You can also 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
```

**Related Information**

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

### 3.2.5. 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 8. Specifying the Quartus Key File

Specify Quartus Key File

Security Category

Set Quartus Key File

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

```plaintext
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.

Figure 9. Programming File Generator

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.
9. 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 7 on page 20 *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.

### 3.2.6. 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

### 3.2.7. 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. eFuses do not use the dedicated V_{CCBAT} power supply. Refer to the Intel Stratix 10 Device Family Pin Connection Guidelines for information about the power supply for writing eFuses.

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.

**Related Information**

Intel Stratix 10 Device Family Pin Connection Guidelines

### 3.2.8. Step 5a: Programming the Owner Root Public Key

2. Right click the image of the Intel Stratix 10 device and select Edit ➤ Add QKY/QEK/Fuse file ....
3. Right-Click the Stratix 10 Device

Add QKY 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**.
— 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.

6. Click **Start** to program the owner root public key.

### 3.2.9. 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**

- eFuses Overview on page 63

### 3.2.10. Verifying a Configuration Bitstream Signature

The Intel Quartus Prime Programming File Generator includes an option to check the integrity of a signed bitstream. This option also verifies the signature blocks in the bitstream and prints important information, such as the expected value of the root key hash, public key values, permissions, and the contents of each signature chain for each section.

The following command runs an integrity check;

```bash
$ quartus_pfg --check_integrity signed_bitstream.rbf
```
Here is an example of the output of the command:

```
Info: Command: quartus_pfg --check_integrity output_file_signed.rbf
Integrity status: OK

Section
Type: CMF
Signature Descriptor ...
Signature chain #0 (entries: 3, offset: 96)
Entry #0
Fuse: A1B9545C CAC4152D 9511A9AB 321778ED 1180A280 6DC58F2C
5607433E 02A872E3 F52B2AE5 F7B8BDE0 53FA000D 8FC7AC04
Generate key ...
Curve : secp384r1
X: FC28C8662DF1437DD98E61336467DC9CDA788F22F949D8F88DA755A9F8CC11AEC10006E2
6490B3EAE8148E6C8AA8A1
Y: 95D1BA0FF4C7374B350FDF39CF66A9E4D516869D49543AC3DB6792273111981058D0E192AD4C075711F6185E1B095DE7F
8D7C6FB62A108A7C512276
Entry #1
Generate key ...
Curve : secp384r1
X: B11534AA67A30E884BF89819281522FD0326BB8F108BC483946717A14F9630C682ECDAE5
40FECB06F3E6B6C921A10A
Y: 0DD5F19E6A38D97148CE6F563B6792273111981058D0E192AD4C075711F6185E1B095DE7F
E2F455851E78F9BF3D2C6
Entry #2
Keychain permission: SIGN_CODE
Keychain can be cancelled by ID: 5
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

Section
Type: IO
Signature Descriptor ...
Signature chain #0 (entries: 5, offset: 96)
Entry #0
Fuse: 46D2D1CD 666F6FA3 8CA6DF11 F09F18E84 41162254 D5E811F0 0B72B678 52D29F2F
Generate key ...
Curve : prime256v1
X: DD4E3FB89EC29E0F2C9435A8D74E0780F2282367EABF4F84FD207A80EFDFA1552
Y: 9A87E4400002AE7F2FF6716FE889C49DD50D4F8BC719532D8267BBF06FF49
Entry #1
Generate key ...
Curve : prime256v1
X: 7E9D2C6D246339E6D58B937D4127F83FF590B64663FEC316A418847AAA8250S
Y: 29E7E71AFC4CDBBB99414C2673EA7AD44B4E4442E803D350590DA0D5A0BF2F5
Entry #2
Generate key ...
Curve : prime256v1
X: 3A9083FF4B911136EA4C3041916C2E1FC887397ABCEA017DE42AF143DBEA17ED8
Y: 4DDDD1670C3F846EFFC4B071BC8D291FD9477EE035AD9C46B96DD2F07502809
Entry #3
Generate key ...
Curve : prime256v1
X: 8A1FBB13D3F05956E7FFF7DE94AFD1836752169A9E66B79B5861BDA79E53F
Y: 361FE17E8C73DE0FB4277480FAED32363AC134D2DD76961E6F046222F06D600
Entry #4
Keychain permission: SIGN_CORE, SIGN_HPS
Keychain can be cancelled by ID: 0, 0, 0
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

Section
Type: HPS
Signature Descriptor ...
```

3. Design Authentication and Signature Chains

Intel® Stratix® 10 Device Security User Guide

30
Signature chain #0 (entries: 5, offset: 96)

Entry #0
Fuse: 46D2D1CD 666F6FA3 8CA6DF11 F09F1E84 41162254 D5E811F0 0B72B678 52D29F2F
Generate key ...
Curve : prime256v1
X: DD4E3FB89EC29E0F2C9435A8D74E0780F2282367EABF4F84FD207A080EFDA1552
Y: 9A8A74E440002AE72FF67716FE889C49DD50FD4FBC7195324DE67BFF066FF49

Entry #1
Generate key ...
Curve : prime256v1
X: 7EF9D2C6D246339E6D58B937D4127F83FF590B64663FEC316A18847AA82505
Y: 29EE71EAF4C0BB99414C2673EA7AD4BE4442E803D350590DA0D95A0F2EF5

Entry #2
Generate key ...
Curve : prime256v1
X: 3A9083FF491136EAC43041916C2E1FC887397ABCEA017DE42AF143DBEA17ED8
Y: 4DDDD1670C3F846EFFC40B71BC8D291FD9477EE035AD9C46B69DD20F5702809

Entry #3
Generate key ...
Curve : prime256v1
X: 8A1FFB3D3F0E5961E7FFF7D8E94ADF1836752169A9E66B79B5B5861BBD97E953F
Y: 361FE17E8C73DE0FB4277480FAED32363A3C134DD27D6961E6F046222F06D600

Entry #4
Keychain permission: SIGN_CORE, SIGN_HPS
Keychain can be cancelled by ID: 0, 0, 0
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

Section
Type: CORE
Signature Descriptor ...
Signature chain #0 (entries: 5, offset: 96)

Entry #0
Fuse: 46D2D1CD 666F6FA3 8CA6DF11 F09F1E84 41162254 D5E811F0 0B72B678 52D29F2F
Generate key ...
Curve : prime256v1
X: DD4E3FB89EC29E0F2C9435A8D74E0780F2282367EABF4F84FD207A080EFDA1552
Y: 9A8A74E440002AE72FF67716FE889C49DD50FD4FBC7195324DE67BFF066FF49

Entry #1
Generate key ...
Curve : prime256v1
X: 7EF9D2C6D246339E6D58B937D4127F83FF590B64663FEC316A18847AA82505
Y: 29EE71EAF4C0BB99414C2673EA7AD4BE4442E803D350590DA0D95A0F2EF5

Entry #2
Generate key ...
Curve : prime256v1
X: 3A9083FF491136EAC43041916C2E1FC887397ABCEA017DE42AF143DBEA17ED8
Y: 4DDDD1670C3F846EFFC40B71BC8D291FD9477EE035AD9C46B69DD20F5702809

Entry #3
Generate key ...
Curve : prime256v1
X: 8A1FFB3D3F0E5961E7FFF7D8E94ADF1836752169A9E66B79B5B5861BBD97E953F
Y: 361FE17E8C73DE0FB4277480FAED32363A3C134DD27D6961E6F046222F06D600
3.3. 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.

3.3.1. 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).

   Note: The Stratix10.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 Intel Stratix 10 device using the signed firmware (Signed FW signed_Stratix10.zip) and owner.fuse as inputs.

   Note: You must power cycle your board after programming the fuses.
3.3.1.1. 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 `--key_storage` argument to specify physical eFuses on the Intel Stratix 10 device. Here are both versions of the `quartus_pgm` command:

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

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

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 27
- Step 1: Creating the Root Key on page 21

3.3.1.2. 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
```

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_root_key.qky
```
3.3.1.3. 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 Stratix10.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 Stratix10.zip signed_Stratix10.zip
```

Refer to Programming eFuses on page 66 for instructions on programming eFuses.

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

After you program the co-signed firmware eFuse, the Intel Stratix 10 device 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_Stratix10.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=1SX280LH2 -o subtype=FUSE 
   -o fw_source=signed_Stratix10.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"
```

3.4. 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.
Table 5.  
Signing Command Argument Summary

<table>
<thead>
<tr>
<th>Argument</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operation</td>
<td>make_private_pem</td>
<td>Generates a private key in .pem format such as root_private.pem.</td>
</tr>
<tr>
<td></td>
<td>make_public_pem</td>
<td>Generates a public key in .pem format from the private .pem such as root_public.pem.</td>
</tr>
<tr>
<td></td>
<td>make_root</td>
<td>Creates a new Intel Quartus Prime keychain .qky file with a given public key .pem in the root entry such as root_public.qky.</td>
</tr>
<tr>
<td></td>
<td>append_key</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>sign</td>
<td>Signs the bitstream with the .pem private key and key chain.</td>
</tr>
<tr>
<td></td>
<td>fuse_info</td>
<td>Calculates the root public key hash from the .qky file.</td>
</tr>
</tbody>
</table>

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.

3.4.1. 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.

3.4.2. 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>
```

continued...
### 3.4.3. 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.

### 3.4.4. 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 `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.
3.4.5. 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.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>quartus_sign --family=stratix10 --operation=sign --qky=&lt;qky file&gt; --pem=&lt;private PEM for the public key of last entry in the input QKY&gt; &lt;unsigned rbf, unsigned firmware zip file, unsigned debug certificate&gt; &lt;signed rbf, firmware zip file, or signed debug certificate&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input file</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sign operation supports the following 3 input file types:</td>
</tr>
<tr>
<td>• unsigned rbf file: This is the .rbf that you generate from the .sof</td>
</tr>
<tr>
<td>• unsigned firmware zip file: quartus/common/devinfo/programmer/firmware/Stratix10.zip. This file contains the SDM firmware.</td>
</tr>
<tr>
<td>• 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 &quot;ei;hps_unsigned.cert;&lt;device&gt;&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed rbf file, signed firmware zip file, or signed certificate file: This file is the output of the sign operation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>This command has 2 additional arguments:</td>
</tr>
<tr>
<td>• qky: This is the .qky file generated by the previous append_key or make_root operation.</td>
</tr>
<tr>
<td>• pem: This is the private PEM for the previous public key of the input QKY.</td>
</tr>
</tbody>
</table>

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

Related Information

Step 4: Signing the Bitstream on page 23

3.4.6. Calculate Root Public Key Hash from QKY

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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>quartus_sign --family=stratix10 --operation=fuse_info &lt;input QKY&gt; &lt;fuse output text&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input file</th>
</tr>
</thead>
<tbody>
<tr>
<td>input QKY: This is the root public key.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuse output text: Manufacturing uses this text file to program the specified eFuses of the Intel Stratix 10 device.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>This command has no additional arguments.</td>
</tr>
</tbody>
</table>
4. Encryption and Decryption

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 quad SPI flash memory as a PUF-wrapped key. 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 11. Security Tab: AES Encryption Key Update Ratio and Enable Scrambling**
Figure 12. **AES Update Mode**

![Diagram showing AES Update Mode]

For \( (N + 1) \), \( N \) = the data block count, \( 1 \) = the key block

**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 13. **Bitstream Decryption**

![Diagram showing Bitstream Decryption]

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. During bitstream generation Intel Quartus Prime Software writes a partially encrypted bitstream to your working directory. The encryption finalization step converts the partially-encrypted bitstream to a fully-encrypted bitstream.
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.

4.1. 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 14. Design Flow for Owner Image Encryption in Intel Stratix 10 Devices

4.1.1. 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)

2. On the Output Files tab, specify the output file type for your configuration scheme.
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.

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.
4.1.2. 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.

4.1.3. 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 17. Specify Storage Location for Encryption Key

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.
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.

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.

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

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 in Physical eFuses on page 45
4.1.4. 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.

   ```shell
   // 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, or Real eFuses.

   **Note:** The current release only supports PUF storage in quad SPI flash memory as part of a .jic file. The following command programs flash memory over the JTAG interface:

   ```shell
   quartus_pgm -c <cable_name> -m jtag -o "p;cfg-bitstream.jic"
   ``

   Here are some additional commonly used arguments to the quartus_pgm command:
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 in Physical eFuses on page 45
- `quartus_pgm` Command Operation Argument on page 79

### 4.1.5. Storing the AES Key in Physical eFuses

Beginning in version 19.3, the Intel Quartus Prime Pro Edition Software supports storing the AES root key in physical eFuses. 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:
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.

### 4.1.6. 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?

### 4.2. Using a PUF-Wrapped AES Key (Beta)

Intel Stratix 10 devices utilize Intrinsic ID technology to implement an SRAM Physically Unclonable Function (PUF) that can be used as a secure and flexible option for AES key storage. A PUF uses unique silicon variations inherent in each Intel Stratix 10 device as a fingerprint that can be used to derive a secret key, called the PUF root key. The PUF root key wraps the AES root key. QSPI flash stores this wrapped key.
The PUF root key is undiscoverable outside of an Intel Stratix 10 device. The process to create the PUF root key, called enrollment, generates a helper data file that is subsequently used in the reconstruction of the root key at each power on. The reconstruction process is called activation. The helper data is also stored in QSPI flash and does not contain any information that could be used to extract the wrapped AES key.

Implementing a PUF-wrapped AES Key includes the following steps:
1. Enrolling the Intrinsic ID PUF via JTAG.
2. Generating programming files for flash programming.
3. Programming the Intrinsic ID PUF data into quad SPI flash memory.

Related Information
Acronyms and Glossary on page 77

4.2.1. Step 1: Enrolling the Intrinsic ID PUF and Wrapping the AES Key

PUF enrollment creates the PUF helper data file (.puf) from unique physical variations in your Intel Stratix 10 device. PUF Activation reads the helper data and derives the PUF root key. The PUF root key unwraps a wrapped AES key. The SDM automatically activates the PUF whenever you power cycle your system if PUF data is available in the quad SPI flash device.

If you believe that the PUF in flash memory is corrupt, you can recreate new helper data by running the command to generate helper data again. New helper data is different from any other helper data. New helper data cannot unwrap a wrapped AES root key that was wrapped with different helper data. To keep your .puf and .wkey in sync, always run the commands to enroll PUF data (.puf) followed by the command to generate the IID PUF wrapped AES key (.wkey).

Figure 20. Intrinsic ID PUF Enrollment

1. Use the Intel Quartus Prime Programmer to trigger PUF enrollment and generate the .puf. In the following command the i argument loads a helper image to load the provision firmware so that the provision firmware can generate wrapper.puf.

```
quartus_pgm -c 1 -m jtag -o "ei;wrapper.puf;1SX280LH2"
```

(3) The use of Intrinsic ID technology requires a separate license agreement with Intrinsic ID. Intel Quartus Prime Pro Edition software restricts selection of a PUF-wrapped AES key storage without the appropriate license.
2. Generate the IID PUF wrapped AES key (.wkey) using the following arguments:
   - The private .pem file used in creating the root public key
   - The .qky root public key
   - The .qek encryption key
   - The initialization vector (iv)

   ```sh
   quartus_pgm -c 1 -m jtag --pem_file=design0_sign_private.pem \
   --qky_file=design0_sign_chain.qky --qek_file=aes.qek \
   --iv=1234567890ABCDEF1234567890ABCDEF -o "ei;aes.wkey;1SX280LH2"
   ```

   **Figure 21. Wrapping the AES Key**

   ![Diagram showing the process of wrapping the AES key]

   Successful activation results in a .wkey file in your working directory.

4.2.2. Step 2: Querying Intrinsic ID PUF Activation Status

Use the `quartus_pgm` command to query the PUF activation status. If activation fails, the Intel Quartus Prime Programmer sends the error status to the SDM.

1. Use the following command to query the activation status:

   ```sh
   quartus_pgm -c 1 -m jtag --status --status_type="CONFIG"
   ```

   Here is sample output from a successful activation:

   ```plaintext
   Response of CONFIG_STATUS
   Device is running in user mode
   00000000 RESPONSE_CODE=OK, LENGTH=6
   00000000 STATE=IDLE
   00000000 Version
   80000000 MSEL=JTAG, nSTATUS=1, nCONFIG=1
   80000002 CONF_DONE=0, INIT_DONE=1, CVF_DONE=0, SEU_ERROR=0,
   PROVISION_FW=1
   00000000 Error location
   00000000 Error details
   Response of PUF_STATUS
   00000000 RESPONSE_CODE=OK, LENGTH=1
   00000400 STATUS=PUF_ACTIVATION_SUCCESS, ERROR_BIT_RATE=4
   ```
4.2.3. Step 3a: Specifying the Output Files for Flash Programming

You should already have completed the following steps:

- Specified your Quartus key file file and turned on Enable programming bitstream encryption on the Assignments ➤ Device ➤ Device and Pin Options ➤ Security menu.
- Specified the Quad SPI Intrinsic ID PUF-wrapped for the Encryption key storage select parameter on the Assignments ➤ Device ➤ Device and Pin Options ➤ Security menu,
- Generated you .sof.

Complete the following steps to specify the output files to program to quad SPI flash memory:

1. On the File menu, click Programming File Generator. On the Output Files tab make the following selections:
   a. For Device Family select Stratix 10.
   b. For Configuration mode select Active Serial x4.
   c. For Output directory browse to your output file directory. This example uses output_files
   d. For Name, specify a name for the programming file to be generated. This example uses output_file
   e. Under Description select the programming files to generate. This example generates the JTAG Indirect configuration File (.jic) for device configuration and the Raw Binary File of Programming Helper Image (.rbf) for device helper image. This example also selects the optional Memory Map File (.map) and Raw Programming Data File (.rpd). The raw programming data file is necessary only if you plan to use a third-party programmer in the future.

Figure 22. Programming File Generator - Output Files Tab - Select JTAG Indirect Configuration
4.2.4. Step 3b: Specifying Input Files for Flash Programming

Use the Programming File Generator to specify the configuration bitstream, authentication, and encryption files to include in the .jic quad SPI flash programming file.

Figure 23. Program Flash Memory with PUF Helper Data and Application Image

1. To specify your configuration bitstream, on the Input Files tab, click Add Bitstream and browse to your .sof.
3. To specify the your wrapped AES key file, click Add Raw Data. Change the Files of type drop-down menu to Quartus Wrapped Key File (*.wkey). Browse to your .wkey file.
4. To add your private .pem file, select your .sof file and then click Properties.
   a. Turn On Enable signing tool.
   b. For Private key file select your .pem file.
   c. Turn On Finalize encryption.
   d. For Encryption key file select your .qek file.

4.2.5. Step 3c: Specifying the Configuration Device for Flash Programming

Use the Programming File Generator to specify your configuration device and generate the files you specified on the Output Files tab.

1. On the Configuration Device tab click Add Device and select your flash device from the list of available flash devices.
2. Select the configuration device you have just added and click Add Partition.
3. In the Edit Partition dialog box for the Input file and select your .sof from the dropdown list. You can retain the defaults or edit the other parameters in the Edit Partition dialog box.
4. When you add the .puf and .wkey as input files, the Programming File Generator automatically creates a PUF partition in your Configuration Device. To store the .puf and .wkey in the PUF partition, select the PUF partition and click Edit. In the Edit Partition dialog box, select your .puf and .wkey files from the drop-down lists.

5. For the Flash Loader parameter select the Intel Stratix 10 device family and device name that matches your Intel Stratix 10 OPN.

6. Click Generate to generate the output files that you specified on the Output Files tab.

7. The Programming File Generator reads your .qek file and prompts you for your passphrase if you generated your AES key with a passphrase. Type your passphrase in response to the Enter QEK passphrase prompt. Click the Enter key.
Type passphrase

Note: As you type your passphrase, you do not see any indication that you are typing characters.

8. Click OK when the Programming File Generator reports successful generation.

4.2.6. Step 4: Programming the PUF-Wrapped Key to Quad SPI Flash

Use the Programmer to write your .jic to quad SPI flash memory. This file includes the following files:

- The configuration bitstream: .sof
- The PUF helper data: .puf
- The PUF-wrapped AES key: .wkey
- The private signing key: .pem
- The Intel Quartus Prime encryption key file: .qek

2. In the Programmer, click Hardware Setup and then select a connected Intel FPGA Download Cable.
3. Click Add File and browse to your .jic file.

Figure 27. Program .jic

4. Turn on Program/Configure to program your quad SPI flash memory. The quad SPI flash memory device subsequently loads the configuration data into the target FPGA via AS x4 configuration.
4.2.7. Enabling and Disabling the More Security Options

The Assignments ➤ Device ➤ Device and Pin Options ➤ Security ➤ More options dialog box provides additional security options. Several of these parameters address the location of the encryption key.

When you turn on any of these options, the Intel Quartus Prime Software sets a flag in the .sof that the SDM reads before configuring the Intel Stratix 10 device. These options replicate most of the functionality that you can control using eFuses. However, programming physical eFuses makes these options permanent. Setting these options in the More Security Options dialog box allows you to turn them off and on by recompiling your design.

Complete the following steps specify these additional security settings:

2. In the More Security Options dialog, turn on the options you want to disable.

Figure 28. Specifying Security eFuse Settings

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable JTAG</td>
<td>Off</td>
</tr>
<tr>
<td>Force SDM clock to internal oscillator</td>
<td>Off</td>
</tr>
<tr>
<td>Force encryption key update</td>
<td>Off</td>
</tr>
<tr>
<td>Disable virtual eFuses</td>
<td>Off</td>
</tr>
<tr>
<td>Lock security eFuses</td>
<td>Off</td>
</tr>
<tr>
<td>Disable HPS debug</td>
<td>Off</td>
</tr>
<tr>
<td>Disable encryption key in eFuses</td>
<td>Off</td>
</tr>
<tr>
<td>Disable encryption key in BRAM</td>
<td>Off</td>
</tr>
<tr>
<td>Disable Intrinsic ID PUF-wrapped encryption key in Quad SPI</td>
<td>Off</td>
</tr>
</tbody>
</table>

Description:
When set, the device does not use a PUF-wrapped AES key stored in Quad SPI. Disabling the storage locations that you are not using may prevent an attack that uses a different storage location for the encryption key.

Related Information
Programming eFuses on page 66
4.2.8. Additional Information About Using the PUF

4.2.8.1. Location of the PUF in Flash Memory

The location of the PUF file is different for designs that support RSU and designs that do not support the RSU feature.

Table 6. Flash Sub-Partitions Layout without RSU Support

<table>
<thead>
<tr>
<th>Flash Offset</th>
<th>Size (in bytes)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 K</td>
<td>256 K</td>
<td>Configuration Management Firmware</td>
<td>Firmware that runs on SDM.</td>
</tr>
<tr>
<td>256 K</td>
<td>256 K</td>
<td>Configuration Management Firmware</td>
<td></td>
</tr>
<tr>
<td>512 K</td>
<td>256 K</td>
<td>Configuration Management Firmware</td>
<td></td>
</tr>
<tr>
<td>768 K</td>
<td>256 K</td>
<td>Configuration Management Firmware</td>
<td></td>
</tr>
<tr>
<td>1M</td>
<td>32 K</td>
<td>PUF data copy 0</td>
<td>Data structure for storing PUF helper data and PUF-wrapped AES root key copy 0</td>
</tr>
<tr>
<td>1M+32 K</td>
<td>32 K</td>
<td>PUF data copy 1</td>
<td>Data structure for storing PUF helper data and PUF-wrapped AES root key copy 1</td>
</tr>
</tbody>
</table>

Table 7. Flash Sub-Partitions Layout with RSU Support

<table>
<thead>
<tr>
<th>Flash Offset</th>
<th>Size (in bytes)</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 K</td>
<td>256 K</td>
<td>Decision firmware</td>
<td>Firmware to identify and load the highest priority image.</td>
</tr>
<tr>
<td>256 K</td>
<td>256 K</td>
<td>Decision firmware</td>
<td></td>
</tr>
<tr>
<td>512 K</td>
<td>256 K</td>
<td>Decision firmware</td>
<td></td>
</tr>
<tr>
<td>768 K</td>
<td>256 K</td>
<td>Decision firmware</td>
<td></td>
</tr>
<tr>
<td>1M</td>
<td>8 K + 24K Padding</td>
<td>Decision firmware data</td>
<td>Reserved for Decision firmware use.</td>
</tr>
<tr>
<td>1M + 32 K</td>
<td>Variable</td>
<td>Factory image</td>
<td>A simple image that you create as a backup if all other application images fail to load. This image includes the CMF that runs on the SDM.</td>
</tr>
<tr>
<td>Next</td>
<td>32 K</td>
<td>PUF data copy 0</td>
<td>Data structure for storing PUF helper data and PUF-wrapped AES root key copy 0</td>
</tr>
<tr>
<td>Next +32 K</td>
<td>32 K</td>
<td>PUF data copy 1</td>
<td>Data structure for storing PUF helper data and PUF-wrapped AES root key copy 1</td>
</tr>
<tr>
<td>Next + 256 K</td>
<td>4 K</td>
<td>Sub-partition table copy 0</td>
<td>Data structure to facilitate the management of the flash storage.</td>
</tr>
<tr>
<td>Next +32 K</td>
<td>4 K</td>
<td>Sub-partition table copy 1</td>
<td></td>
</tr>
<tr>
<td>Next +32 K</td>
<td>4 K</td>
<td>CMF pointer block copy 0</td>
<td>A list of pointers to application images in order of priority. When you add an image, that image becomes the highest.</td>
</tr>
<tr>
<td>Next +32 K</td>
<td>CMF pointer block copy 1</td>
<td>A second copy of the list of pointers to application images.</td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Variable</td>
<td>Application image 1</td>
<td>Your first application image.</td>
</tr>
<tr>
<td>Variable</td>
<td>Variable</td>
<td>Application image 1</td>
<td>Your second application image.</td>
</tr>
</tbody>
</table>
4.2.8.2. Intrinsic ID PUF Re-Enrollment

You can update the PUF data in flash after enrolling a new Intrinsic ID PUF (.puf) or updating the PUF-wrapped AES key, (.wkey). In the current release, you use the JTAG interface to erase the .jic and reprogram the entire flash device.

4.3. Encryption Command Detailed Description

The encryption command supports the following functions:
- Making an AES key
- Encrypting the configuration bitstream

Table 8. Signing Command Argument Summary

<table>
<thead>
<tr>
<th>Argument</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>encrypt</td>
<td>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.</td>
</tr>
</tbody>
</table>

4.3.1. Make AES Key

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

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>quartus_encrypt --family=stratix10 --operation=make_aes_key &lt;output .qek&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input file</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output file</td>
<td>.qek</td>
</tr>
</tbody>
</table>

**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:

\[ \log_2(\text{max_key_use}) \times (\text{ik_count} + 1) < 64 \]

- **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.
4.3.2. 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.

<table>
<thead>
<tr>
<th>Command</th>
<th><code>quartus_encrypt --family=stratix10 --operation=encrypt --Key=&lt;output .qek&gt; \ &lt;partially-encrypted .rbf&gt; &lt;fully encrypted .rbf&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Input file</td>
<td>Partially encrypted .rbf file.</td>
</tr>
<tr>
<td>Output file</td>
<td>Fully encrypted .rbf file.</td>
</tr>
</tbody>
</table>
| Arguments | This command includes the following 2 required arguments:  
  • `key`: Specifies a .qek file.  
  • `passphrase`: Specifies an optional file path that contains passphrase to protect the .qek. If you do not specify this argument, the `quartus_encrypt` command prompts you to enter the passphrase.
5. Anti-Tamper Monitoring and Mitigation (Beta)

Adversaries can use tampering attacks to alter the physical conditions or inputs to a system to cause unintended behavior or reverse-engineer the system capabilities. The Intel Stratix 10 device family includes robust anti-tampering features to help detect, respond to, and protect against such attacks.

Intel Stratix 10 devices employ both passive and active anti-tamper features. Active anti-tamper features allow you to choose the detection thresholds and responses to meet your threat model and system requirements.

Passive anti-tamper features run continuously and do not have customizable detection or response behaviors. These anti-tamper features impede attempts to influence or reverse-engineer device behavior. Intel Stratix 10 devices include the following passive anti-tamper features:

- Lock-step operation of three SDM processors for device configuration and subsequent operations
- Redundant eFuse settings
- Extensive integrity protections in the configuration bitstream

Intel Stratix 10 devices also support active monitoring of operating conditions including certain input clocks, voltage rails, and the device temperature. You can specify the monitoring thresholds for operating conditions that may indicate an attempt to tamper with the device. You can also specify the SDM response level when a monitor detects a tampering event. You specify anti-tamper settings as part of the design process. The configuration bitstream includes this information. Consequently, anti-tamper features are active during configuration and remain active until you load a new configuration bitstream.

Enabling active anti-tamper features requires three steps:

- Specifying the optional SDM I/O pins to monitor anti-tampering status
- Specifying the detection parameters
- Specifying the desired response.

5.1. Specifying Anti-Tamper Pins

Intel Quartus Prime Software provides the following two optional anti-tamper SDM I/O pins.

- **TAMPERDETECTION**: The SDM drives the TAMPERDETECTION pin high when the value of any monitored feature exceeds thresholds you specify.
- **TAMPERZEROIZATIONSTATUS**: The SDM drives the TAMPERZEROIZATIONSTATUS pin high to confirm a successful response.
Once set, the anti-tamper pins remain active until the next power-on reset. Complete the following steps to enable the anti-tamper SDM I/O pins:

1. On the Assignments ➤ Device ➤ Device and Pin Options ➤ Configuration Pin Options page enable the Tamper detect output and the Anti-tamper response status output options.

2. For each anti-tamper pin, select an unused pin from the dropdown list. If you select an SDM I/O that is already in use, the pin color changes to red.

![Figure 29. Enabling the Tamper Detection and Status SDM I/O Signals and Assigning Pins](image)

5.2. Specifying the Anti-Tamper Detection Parameters

The SDM monitors the frequency of the external configuration reference clock, osc clk 1, the \( V_{cc1} \) and \( V_{cc2}_{SDM} \) voltage rails, and the temperature of the device. If the Intel Stratix 10 device exceeds the threshold that you specify, the threshold monitor triggers a tamper response.

Complete the following steps to specify thresholds for the reference clock, voltage rails, and device temperature.
1. On the Assignments ➤ Device ➤ Device and Pin Options ➤ Security ➤ Anti-Tamper tab, specify the Anti-tamper response for tampering events. Refer to Table 9 on page 61 for descriptions of each response.

2. For Enable frequency tamper detection select the percent deviation in frequency from the Frequency tamper detection range list.

3. For Enable temperature tamper detection specify a Temperature tamper upper bound and Temperature tamper lower bound. Refer to the Intel Stratix 10 Device Datasheet Absolute Maximum Ratings, Recommended Operating Conditions and External Temperature Sensing Diode Specifications for information about supported temperature ranges.

4. Under Voltage detection complete the following steps;
   a. Turn on Enable VCCL voltage tamper detection and Enable VCCL SDM tamper detection to monitor these voltages.
   b. For Voltage tamper detection trigger specify a percentage deviation from the expected nominal voltage range to trigger a tamper response.
Figure 30. Anti-Tamper Parameters

Note: In addition to the anti-tamper features that Intel provides, you can also implement other anti-tamper features in the Intel Stratix 10 device fabric.
5.3. Understanding the Anti-Tamper Responses

When a monitored feature exceeds the threshold you specify, the SDM initiates the response that you specify. The SDM puts the Intel Stratix 10 device in a stopped state that requires a power-on reset to resume operation.

The Intel Quartus Prime Software Release 20.1 supports two response levels.

Table 9. Anti-Tamper Response Level Descriptions

<table>
<thead>
<tr>
<th>Anti-Tamper Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Cleaning, Zeroization, BBRAM key cleaning</td>
<td>The SDM completes the following actions:</td>
</tr>
<tr>
<td></td>
<td>• Asserts the TAMPERDETECTION SDM I/O pin, if you have enabled this pin.</td>
</tr>
<tr>
<td></td>
<td>• Runs a series of operations to clear the configuration information from the FPGA. The SDM also clears information from the HPS on-chip RAM.</td>
</tr>
<tr>
<td></td>
<td>• Stops any running operations and clears all resources in use such as the key vault.</td>
</tr>
<tr>
<td></td>
<td>• Drives the TAMPERZEROIZATIONSTATUS SDM I/O pin high when zeroization operations complete successfully. This action only occurs if you enable the TAMPERZEROIZATIONSTATUS SDM I/O pin.</td>
</tr>
<tr>
<td></td>
<td>• Disables all SDM security features.</td>
</tr>
<tr>
<td></td>
<td>• Drives the nStatus pin low indicating that the device is currently cleaning and performing zeroization.</td>
</tr>
<tr>
<td></td>
<td>• The SDM instructs the memory controllers to initiate zeroization. Refer to Understanding the Zeroization Tamper Response for more information about zeroization.</td>
</tr>
<tr>
<td></td>
<td>• Clears BBRAM registers and zeroizes the BBRAM key. If you store your AES key in BBRAM, you must reprovision the AES key before you can configure the device with an encrypted bitstream.</td>
</tr>
<tr>
<td></td>
<td>• Disables all SDM activity by disabling both data and control clocks.</td>
</tr>
<tr>
<td>Disabled</td>
<td>The SDM does not respond to tampering events. If you have enabled the TAMPERDETECTION SDM I/O output pin, this pin asserts even if the SDM does not respond.</td>
</tr>
</tbody>
</table>

Related Information
Understanding the Zeroization Tamper Response on page 61

5.4. Understanding the Zeroization Tamper Response

Zeroization clears a memory to the reset state and reads back that memory to confirm all data is clear.

The SDM processors manage zeroization of memories by sending zeroization commands to memory controllers and recording zeroization status. This procedure allows the SDM to perform zeroization even on memories it cannot directly read, such as FPGA configuration memory or the key vault.

The memory controllers zeroize in the following memories in the Intel Stratix 10 device:

- FPGA configuration RAM and embedded RAM. These memories contain the FPGA configuration information.
- HPS on-chip RAM.
The memory controllers clear but do not zeroize the following memories:

- FPGA Intel Hyperflex™ registers.
- DSP and M20K registers
- Configuration network on a chip (NoC) and LSM registers
- Key value BBRAM
6. Using eFuses

6.1. eFuses Overview

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.

Accessing the Virtual eFuses

The SDM maintains a virtual eFuse cache in volatile RAM. The virtual eFuse cache stores values that represent the current state of the physical eFuses, including pending write values and other modified state.

When you use the `quartus_pgm` command to read an eFuse, the Programmer examines the virtual eFuse value in the eFuse cache. When you use the `quartus_pgm` command to write an eFuse value the `key_storage` argument specifies Virtual eFuses or Real eFuses.

You can use Intel Quartus Prime Programmer to write values to the fuse cache in SDM RAM. These values persist until you remove power from the device. Because programming incorrect values into real eFuses renders the device unusable, Intel recommends that you test eFuse programming using virtual eFuses before programming the first Intel Stratix 10 device with real eFuses. Intel recommends using test values when programming virtual eFuses.

The following table describes all available owner eFuses.

<table>
<thead>
<tr>
<th>eFuse Name</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel FPGA public key hash</td>
<td>384-bit hex</td>
<td>Read-only.</td>
</tr>
<tr>
<td>Intel FPGA public key cancellation</td>
<td>32-bit hex</td>
<td>32 Intel cancellation IDs are available. Each bit corresponds to the cancellation ID.</td>
</tr>
<tr>
<td>Co-signed firmware</td>
<td>1-bit boolean</td>
<td>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.</td>
</tr>
<tr>
<td>Device not secure</td>
<td>1-bit boolean</td>
<td>If you receive a device and this fuse is programmed do not use the device and contact Intel.</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>eFuse Name</th>
<th>Legal Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner encryption key program done</td>
<td>1-bit boolean</td>
<td>The Programmer programs the owner AES key into eFuses.</td>
</tr>
<tr>
<td>Owner encryption key program start</td>
<td>1-bit boolean</td>
<td></td>
</tr>
<tr>
<td>Owner key cancellation</td>
<td>32-bit hex</td>
<td>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.</td>
</tr>
<tr>
<td>Owner root public key hash</td>
<td>384-bit hex</td>
<td>Stores the hash value of the owner root public key.</td>
</tr>
<tr>
<td>Owner public key size</td>
<td>[0, 256, 384]</td>
<td>Specifies owner public key size. Intel recommends using 384-bit keys if possible.</td>
</tr>
<tr>
<td>JTAG disable</td>
<td>1-bit boolean</td>
<td>When set, disables JTAG command and configuration. Setting this eFuse eliminates JTAG as mode of attack.</td>
</tr>
<tr>
<td>Force SDM clock to Internal Oscillator</td>
<td>1-bit boolean</td>
<td>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.</td>
</tr>
<tr>
<td>Force encryption key update</td>
<td>1-bit boolean</td>
<td>When set, all encrypted bitstreams must specify the Encryption Update Ratio on the Intel Quartus Prime Assignments ➤ Device and Pin Options ➤ Security menu.</td>
</tr>
<tr>
<td>Disable virtual eFuses</td>
<td>1-bit boolean</td>
<td>When set, disables the eFuse virtual programming capability.</td>
</tr>
<tr>
<td>Lock security eFuses</td>
<td>1-bit boolean</td>
<td>Programming this fuse prevents the future programming of any owner-accessible security policy fuses, not including key cancellation ID fuses.</td>
</tr>
<tr>
<td>Disable HPS debug</td>
<td>1-bit boolean</td>
<td>When set, permanently disables debugging using JTAG to access the HPS.</td>
</tr>
<tr>
<td>Disable encryption key in eFuses</td>
<td>1-bit boolean</td>
<td>When set, the device does not use an AES key stored in eFuses.</td>
</tr>
<tr>
<td>Disable encryption key in BBRAM key disable</td>
<td>1-bit boolean</td>
<td>When set, the device does not use AES key stored in BBRAM.</td>
</tr>
<tr>
<td>Disable PUF-wrapped encryption key</td>
<td>1-bit boolean</td>
<td>When set, the device does not use a PUF-wrapped AES key.</td>
</tr>
<tr>
<td>Disable Intrinsic ID PUF enablement</td>
<td>1-bit boolean</td>
<td>When set, Intrinsic ID PUF re-enrollment fails. Disabling re-enrollment prevents further usage of the PUF in this device if the helper data is corrupted for any reason.</td>
</tr>
</tbody>
</table>

**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 virtual eFuses. Then, you compare that value to the value that the **Examine** function stores in `hashFuse.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 `hashFuse.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 29*

### 6.1.1. 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
6.1.2. Understanding 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. In general, a fuse with a value of 0x0 is not programmed. A fuse with the value of 0xF is programmed.

6.1.3. Programming eFuses

You can program eFuses to enable or disable device security features, improving the security of your design. 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 31. 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=1SX280LH2 -o subtype=FUSE \
   -o fw_source=Stratix10.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.
6.1.4. 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:

   ```bash
   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                          = "0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000"
   # Owner key cancellation               = ""
   # Owner public key hash                = "0x000000000000000000000000000000000CE520B15B082E67ACEBCB8545CE239FDBB8CDE6083F6DF9D3BF542932EA5039"
   # 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:

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

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

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

You can use the `quartus_pfg` command-line command to convert .qky, .qek, and .fuse files to Jam 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 do this by provisioning the AES key in a secure environment.

**quartus_pfg Conversion Commands**

Here are examples of quartus_pfg conversion commands:

```bash
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:

```bash
// 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
quartus_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
7. Additional Security Features

7.1. JTAG Disable

The Disable JTAG on the Assignments ➤ Device ➤ Device and Pin Options ➤ Security ➤ More Options page disables JTAG communication with the SDM. When you set this option, the device does not respond to JTAG commands or configure via JTAG.

This setting does not affect mandatory JTAG instructions such as boundary-scan and reading the ID code. You can reinforce this option by programming the Disable JTAG eFuse.

**Caution:** Programming the Disable JTAG eFuse permanently disables communication between the SDM and JTAG.

7.2. Using an HPS Debug 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 32. Specify Either HPS or SDM Pins for the HPS DAP**

- 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 21
- **eFuses Overview** on page 63

### 7.3. 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 21.
2. **Step 3: Appending the Design Signature Key to the Signature Chain** on page 22.
   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.

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

3. Sign the HPS debug certificate using the `quartus_sign` command:
   ```sh
   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.
   ```sh
   quartus_pgm -c 1 -m jtag -o "p;signed_hps_debug.cert"
   ```
# Intel Stratix 10 Device Security User Guide Archives

<table>
<thead>
<tr>
<th>Intel Quartus Prime Version</th>
<th>User Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.3</td>
<td>Intel Stratix 10 Device Security User Guide Archive</td>
</tr>
<tr>
<td>19.1</td>
<td>Intel Stratix 10 Device Security User Guide Archive</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020.04.13</td>
<td>20.1</td>
<td>Made the following changes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added topic: Important Notice to Customers Regarding Features Added in Intel Quartus Prime Pro Edition Software Version 20.1. It states that the IID PUF-based AES key storage and Anti-Tamper features are a beta release in Intel Quartus Prime Pro Edition software version 20.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for a PUF-wrapped AES key. Refer to Using a PUF-Wrapped AES Key (Beta) for more information. This feature is a beta release in Intel Quartus Prime Pro Edition software version 20.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for monitors that can trigger an anti-tamper response when the temperature, voltage, or external clock frequency exceeds the values you specify. Refer to Anti-Tamper Monitoring and Mitigation (Beta) for more information. This feature is a beta release in Intel Quartus Prime Pro Edition software version 20.1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added the Comparison of AES Key Storage Options table that describes the features of each of the 4 possible storage locations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added firmware ID=6 along with its firmware release in the Intel Firmware IDs table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reorganized user guide.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for a quartus_pfg command that checks the integrity of a signed configuration bitstream. Refer to Verifying a Configuration Bitstream Signature for more information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added an appendix covering acronyms and definitions of security terminology.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added an appendix describing file types that implement security features.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added an appendix showing help for the operation (-o) argument to the quartus_pgm command.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Security Category figures to show the new Permitted owner cancellation id and Anti-Tamper tab.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Removed statement that the JTAG disable eFuse eliminates boundary scan. In the Intel Quartus Prime Release 20.1 release, disabling JTAG does not disable boundary scan.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected minor errors and spelling mistakes.</td>
</tr>
<tr>
<td>2020.01.15</td>
<td>19.3</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>quartus_pfg -c encryption_enabled.sof top.rbf \</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-o finalize_encryption=ON -o qek_file=aes.qek \</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-o signing=ON -o pem_file=design0_sign_private.pem</td>
</tr>
<tr>
<td>2020.01.06</td>
<td>19.3</td>
<td>Made the following changes:</td>
</tr>
</tbody>
</table>

*Other names and brands may be claimed as the property of others.*
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>• Corrected the <code>quartus_encrypt</code> command in the Step 1: Preparing the Owner Image and AES Key File topic. The <code>ik_count</code> and <code>max_key_use</code> arguments must be preceded by <code>--</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added command showing how to convert an <code>.rbf</code> to <code>.jam</code> format in the Step 4: Signing the Bitstream topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added the following note to the Converting Key, Encryption, and Fuse Files to Jam Staple File Formats topic:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Caution:</strong> When you convert the AES <code>.qek</code> file to <code>.jam</code> format, the <code>.jam</code> file contains the AES key in plaintext but obfuscated form. Consequently, you must protect the <code>.jam</code> file when storing the AES key. You can protect the <code>.jam</code> file by provisioning the AES key in a secure environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td>2019.10.30</td>
<td>19.3</td>
<td>Added the following new security features:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for physical (non-volatile) eFuses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Changed the way you specify virtual (volatile) or physical (non-volatile) eFuses. The <code>--non_volatile_key</code> parameter is now an argument to the <code>quartus_pgm</code> command. Consequently, you no longer need to recompile to change the eFuse storage location.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Increased the number of public keys entries supported from 2 to 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for a signed secure HPS debug certificate to prevent unauthorized remote or physical access to the HPS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Decreased the encryption update ratio from 127:1 to 31:1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added support for 10 additional eFuses described in the Owner Programmable eFuses table.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added examples of advanced security features.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added descriptions of side-channel mitigation features.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added the following topics:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Step 4a: Protecting the AES Key when Storing the AES in eFuses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Step 4b: Protecting the AES Key when Storing the AES Key in BBRAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Encryption Command Detailed Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Make AES Key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Encrypt the Bitstream</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Programming eFuses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Canceling eFuses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added examples of <code>.jam</code> commands under the Using the <code>.jam</code> Files to Program Root Key and AES Encryption Key heading.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected AES Update Mode figure. The number of data bits in a data block is 256, not 128.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected the cancellation ID Numbers in Figure 5: Three-Key Signature Chain. The cancellation IDs are 0 and 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Revised Anti-Tampering topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Revised the Using eFuses topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected minor errors and typos.</td>
</tr>
</tbody>
</table>

continued...
<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 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. |
A. Appendices

A.1. Acronyms and Glossary

Table 11. Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Activation Code</td>
<td>Consists of 2 Helper (firmware) data. Each copy is 32 KB for a total of 64 KB. The SDM retrieves the PUF AES key from this data.</td>
</tr>
<tr>
<td>CMF</td>
<td>Configuration Management Firmware</td>
<td>Firmware for the Secure Device Manager (SDM). The SDM manages device configuration. The CMF implements many functions including the functions listed here:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• FPGA configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Voltage regulator configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Temperature measurement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HPS software load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• HPS Reset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Remote System Update (RSU)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Read, erase, and program flash memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Device security, including authentication and encryption</td>
</tr>
<tr>
<td>CA</td>
<td>Certificate Authority</td>
<td>A organization that validates entities such as configuration bitstreams and binds them to cryptographic keys.</td>
</tr>
<tr>
<td>Decision CMF</td>
<td>Decision Configuration Management Firmware</td>
<td>Firmware to identify and load the highest priority image.</td>
</tr>
<tr>
<td>Device Owner</td>
<td>—</td>
<td>The device owner asserts control over a device by programming the root public key into eFuses. When a device has an owner the device only loads configuration bitstreams that the owner has signed.</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
<td>A short piece of added information to help guarantee the integrity and authenticity of a larger piece of information. For example, the SDM uses a MAC to verify PUF helper data during PUF activation.</td>
</tr>
<tr>
<td>Provision Firmware</td>
<td>—</td>
<td>Provision firmware contains code that runs during provisioning. For example provision firmware includes code to program authentication fuses or enroll the PUF.</td>
</tr>
<tr>
<td>PUF</td>
<td>Physically Unclonable Function</td>
<td>A physically-defined digital fingerprint that serves as a unique identifier for a semiconductor device such as a microprocessor. This digital fingerprint is based on unique physical variations which occur naturally during semiconductor manufacturing.</td>
</tr>
</tbody>
</table>

*Other names and brands may be claimed as the property of others.
<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUF helper data</td>
<td>.puf</td>
<td>A binary file that contains PUF helper data. The SDM uses this data to activate the PUF prior to configuration. A 32 kilobyte block in Quad SPI memory stores the PUF helper data.</td>
</tr>
<tr>
<td>UID</td>
<td>Unique ID</td>
<td>An identifier that is guaranteed to be unique.</td>
</tr>
<tr>
<td>Wrapped key file</td>
<td>.wkey</td>
<td>A binary file that contains the PUF-wrapped encryption key.</td>
</tr>
</tbody>
</table>

### A.2. File Types for Security

The Programming File Generator, Intel Quartus Prime Programmer, and `quartus_pgm` command use these file types to generate and program a secure configuration bitstream. Refer to the *Secondary Programming Files (Programming File Generator)* in the *Intel Quartus Prime Pro Edition User Guide: Programmer* for definitions of other configuration bitstream file types such as `.rbf` and `.jic` that are do not implement security-related functions.

#### Table 12. File Types for Security

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.fuse</td>
<td>Contains device eFuse data.</td>
</tr>
<tr>
<td>.pem</td>
<td>Privacy enhanced mail certification. Stores the public or private key for authentication.</td>
</tr>
<tr>
<td>.puf</td>
<td>A binary file that contains PUF helper data. The SDM uses this data to generate the physically unclonable function (.puf) file.</td>
</tr>
<tr>
<td>.qek</td>
<td>Intel Quartus Prime encryption key. A password-protected file that stores the AES key that encrypts and decrypts the configuration bitstream.</td>
</tr>
<tr>
<td>.qky</td>
<td>Intel Quartus Prime key file. Stores the signing key chain for configuration bitstream authentication.</td>
</tr>
<tr>
<td>.wkey</td>
<td>Wrapped AES key file. A binary file that contains the PUF-wrapped encryption key.</td>
</tr>
</tbody>
</table>

### Related Information

Intel Quartus Prime Pro Edition Secondary Programming Files (Programming File Generator)
A.3. quartus_pgm Command Operation Argument

Here is the Intel Quartus Prime help for the operation (-o) argument to the quartus_pgm command.

Quartus_pgm --help=operation

JTAG Config/Program
-o p;file.sof
  -o pvb;file.pof
  -o p;file.rbf
  -o pvbi;file.jic
  -o pvbi;file.qky
  -o pbi;file.gek
  -o p;file.cert

JTAG Examine
-o e;file.pof;device_name
  -o ei;file.jic;device_name
  -o ei;file.fuse;device_name
  -o ei;file.cert;device_name

Skip Device (JTAG Bypass)
-o s;device_name

Passive Serial Program
-o file.sof

Active Serial Program
-o p;file.pof

Passive Serial Chain
-o file1.sof -o file2.sof -o file3.sof

JTAG Chain
-o p;file1.pof -o s;file2.pof
  -o v;file1.pof@1 -o p;file2.pof@2

CDF
quartus_pgm -c byteblastermv[lpt1] file.cdf