Table of Contents

Revision History..................................................................................................................6

Chapter 1: Introduction........................................................................................................ 7
  Installing Bootgen........................................................................................................... 8
  Boot Time Security........................................................................................................ 8

Chapter 2: Boot Image Layout..........................................................................................10
  Zynq-7000 SoC Boot and Configuration....................................................................... 10
  Zynq UltraScale+ MPSoC Boot and Configuration....................................................... 19

Chapter 3: Creating Boot Images.....................................................................................32
  Boot Image Format (BIF)............................................................................................ 32
  BIF Syntax and Supported File Types......................................................................... 33
  Attributes .................................................................................................................. 35

Chapter 4: Using Bootgen Interfaces...............................................................................40
  Bootgen GUI Options................................................................................................... 40
  Using Bootgen on the Command Line......................................................................... 41
  Commands and Descriptions....................................................................................... 42

Chapter 5: Boot Time Security.......................................................................................45
  Using Encryption......................................................................................................... 46
  Using Authentication................................................................................................... 54
  Using HSM Mode........................................................................................................ 64

Chapter 6: FPGA Support...............................................................................................81
  Encryption and Authentication..................................................................................... 81
  HSM Mode.................................................................................................................. 82

Appendix A: Use Cases and Examples...........................................................................85
  Zynq MPSoc Use Cases............................................................................................... 85

Appendix B: BIF Attribute Reference..............................................................................95
spksignature............................................................................................................................ 122
spk_select................................................................................................................................. 122
sskfile........................................................................................................................................ 123
startup...................................................................................................................................... 124
trustzone.................................................................................................................................. 124
udf_bh....................................................................................................................................... 125
udf_data....................................................................................................................................126
xip_mode.................................................................................................................................. 126

Appendix C: Command Reference.................................................................128
arch........................................................................................................................................... 128
bif_help..................................................................................................................................... 128
dual_qspi_mode.......................................................................................................................129
efuseppkbits............................................................................................................................ 129
encrypt......................................................................................................................................130
enryption_dump......................................................................................................................... 130
fill...............................................................................................................................................131
generate_hashes..................................................................................................................... 131
generate_keys........................................................................................................................ 132
image........................................................................................................................................ 134
log............................................................................................................................................. 134
nonbooting.............................................................................................................................. 135
o.................................................................................................................................................135
p.................................................................................................................................................136
padimageheader.....................................................................................................................136
process_bitstream................................................................................................................... 137
read........................................................................................................................................... 137
spksignature............................................................................................................................ 138
split............................................................................................................................................138
verify......................................................................................................................................... 139
verify_kdf.................................................................................................................................. 139
w................................................................................................................................................ 140
zynqmpes1............................................................................................................................... 140
Initialization Pairs and INT File Attribute............................................................................. 141

Appendix D: Bootgen Utility............................................................................143

Appendix E: Additional Resources and Legal Notices..............................145
Documentation Navigator and Design Hubs......................................................... 145
## Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10/30/2019 Version 2019.2</strong></td>
<td></td>
</tr>
<tr>
<td>General updates</td>
<td>Minor editorial changes.</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Added Zynq UltraScale+ MPSoC Secure Header</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Updated Creating a Zynq-7000 SoC Device Boot Image using HSM Mode.</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Added attributes:</td>
</tr>
<tr>
<td></td>
<td>• aarch32_mode</td>
</tr>
<tr>
<td></td>
<td>• big_endian</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Added command <code>verify_kdf</code></td>
</tr>
<tr>
<td><strong>05/22/2019 Version 2019.1</strong></td>
<td></td>
</tr>
<tr>
<td>General updates</td>
<td>Minor editorial changes.</td>
</tr>
</tbody>
</table>
Introduction

Xilinx® FPGAs and system-on-chip (SoC) devices typically have multiple hardware and software binaries used to boot them to function as designed and expected. These binaries can include FPGA bitstreams, firmware images, bootloaders, operating systems, and user-chosen applications that can be loaded in both non-secure and secure methods.

Bootgen is a Xilinx tool that lets you stitch binary files together and generate device boot images. Bootgen defines multiple properties, attributes and parameters that are input while creating boot images for use in a Xilinx device.

The secure boot feature for Xilinx devices uses public and private key cryptographic algorithms. Bootgen provides assignment of specific destination memory addresses and alignment requirements for each partition. It also supports encryption and authentication, described in Using Encryption and Using Authentication. More advanced authentication flows and key management options are discussed in Using HSM Mode, where Bootgen can output intermediate hash files that can be signed offline using private keys to sign the authentication certificates included in the boot image. The program assembles a boot image by adding header blocks to a list of partitions. Optionally, each partition can be encrypted and authenticated with Bootgen. The output is a single file that can be directly programmed into the boot flash memory of the system. Various input files can be generated by the tool to support authentication and encryption as well. See BIF Syntax and Supported File Types for more information.

Bootgen comes with both a GUI interface and a command line option. The tool is integrated into the software development toolkit, Vitis™ Integrated Development Environment (IDE), for generating basic boot images using a GUI, but the majority of Bootgen options are command line-driven. Command line options can be scripted. The Bootgen tool is driven by a boot image format (BIF) configuration file, with a file extension of *.bif. Along with Xilinx SoC, Bootgen has the ability to encrypt and authenticate partitions for Xilinx 7 series and later FPGAs, as described in FPGA Support. In addition to the supported command and attributes that define the behavior of a Boot Image, there are utilities that help you work with Bootgen. Bootgen code is now available on Github.
Installing Bootgen

You can use Bootgen in GUI mode for simple boot image creation, or in a command line mode for more complex boot images. The command line mode commands can be scripted too. You can install Bootgen from Vivado Design Suite Installer or standalone. Vitis is available for use when you install the Vivado® Design Suite, or it is downloaded and installed individually. See the Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973) for all possible installation options.

To install Bootgen from Vivado, go to the Xilinx Download Site, and select the Vivado self-extracting installer. During Vivado installation, choose the option to install Vitis as well. Bootgen is included along with Vitis. You can also install Bootgen from the Vitis Installer. The Vitis self-extracting installer found on the Xilinx Download site. After you install Vitis with Bootgen, you can start and use the tool from the Vitis GUI option that contains the most common actions for rapid development and experimentation, or from the XSCT.

The command line option provides many more options for implementing a boot image. See the Chapter 4: Using Bootgen Interfaces to see the GUI and command line options:

- From the Vitis GUI: See Bootgen GUI Options.
- From the command line using the XSCT option. See the following: Using Bootgen Options on the Command Line.

For more information about Vitis, see Vitis help.

Boot Time Security

Secure booting through latest authentication methods is supported to prevent unauthorized or modified code from being run on Xilinx® devices, and to make sure only authorized programs access the images for loading various encryption techniques.

For device-specific hardware security features, see the following documents:

- Zynq-7000 SoC Technical Reference Manual (UG585)

See Using Encryption and Using Authentication for more information about encrypting and authenticating content when using Bootgen.
The Bootgen hardware security monitor (HSM) mode increases key handling security because the BIF attributes use public rather than private RSA keys. The HSM is a secure key/signature generation device which generates private keys, encrypts partitions using the private key, and provides the public part of the RSA key to Bootgen. The private keys do not leave the HSM. The BIF for Bootgen HSM mode uses public keys and signatures generated by the HSM. See Using HSM Mode for more information.
Chapter 2

Boot Image Layout

This section describes the format of the boot image for different architectures.

- For information about using Bootgen for Zynq-7000 devices, see Zynq-7000 SoC Boot and Configuration.
- For information about using Bootgen for Zynq® UltraScale+™ MPSoC devices, see Zynq UltraScale+ MPSoC Boot and Configuration.
- For information on how to use Bootgen for Xilinx FPGAs, see Chapter 6: FPGA Support.

Building a boot image involves the following steps:

1. Create a BIF file.
2. Run the Bootgen executable to create a binary file.

   Note: For the Quick Emulator (QEMU) you must convert the binary file to an image format corresponding to the boot device.

Each device requires files in a specific format to generate a boot image for that device. The following topics describe the required format of the Boot Header, Image Header, Partition Header, Initialization, and Authentication Certificate Header for each device.

Zynq-7000 SoC Boot and Configuration

This section describes the boot and configuration sequence for Zynq®-7000 SoC. See the Zynq-7000 SoC Technical Reference Manual (UG585) for more details on the available first stage boot loader (FSBL) structures.

BootROM on Zynq-7000 SoC

The BootROM is the first software to run in the application processing unit (APU). BootROM executes on the first Cortex™ processor, A9-0, while the second processor, Cortex, A9-1, executes the wait for event (WFE) instruction. The main tasks of the BootROM are to configure the system, copy the FSBL from the boot device to the on-chip memory (OCM), and then branch the code execution to the OCM.
Optionally, you can execute the FSBL directly from a Quad-SPI or NOR device in a non-secure environment. The master boot device holds one or more boot images. A boot image is made up of the boot header and the first stage boot loader (FSBL). Additionally, a boot image can have programmable logic (PL), a second stage boot loader (SSBL), and an embedded operating system and applications; however, these are not accessed by the BootROM. The BootROM execution flow is affected by the boot mode pin strap settings, the Boot Header, and what it discovers about the system. The BootROM can execute in a secure environment with encrypted FSBL, or a non-secure environment. The supported boot modes are:

- JTAG mode is primarily used for development and debug.
- NAND, parallel NOR, Serial NOR (Quad-SPI), and Secure Digital (SD) flash memories are used for booting the device. The Zynq SoC Technical Reference Manual (UG585) provides the details of these boot modes. See Zynq-7000 Boot and Configuration AR#52538 for answers to common boot and configuration questions.

**Zynq-7000 SoC Boot Image Layout**

The following is a diagram of the components that can be included in a Zynq®-7000 SoC boot image.

![Figure 1: Boot Header](image)

<table>
<thead>
<tr>
<th>Boot Header</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Initialization Table</td>
<td></td>
</tr>
<tr>
<td>Image Header Table</td>
<td></td>
</tr>
<tr>
<td>Image Header 1</td>
<td>Image Header 2</td>
</tr>
<tr>
<td>Partition Header 1</td>
<td>Partition Header 2</td>
</tr>
<tr>
<td>Header Authentication Certificate (Optional)</td>
<td>AC (Optional)</td>
</tr>
<tr>
<td>Partition 1 (FSBL)</td>
<td>AC (Optional)</td>
</tr>
<tr>
<td>Partition 2</td>
<td>AC (Optional)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Partition n</td>
<td>AC (Optional)</td>
</tr>
</tbody>
</table>
Zynq-7000 SoC Boot Header

Bootgen attaches a boot header at the beginning of a boot image. The Boot Header table is a structure that contains information related to booting the primary bootloader, such as the FSBL. There is only one such structure in the entire boot image. This table is parsed by BootROM to get determine where FSBL is stored in flash and where it needs to be loaded in OCM. Some encryption and authentication related parameters are also stored in here. The additional Boot image components are:

- Zynq-7000 SoC Register Initialization Table
- Zynq-7000 SoC Image Header Table
- Zynq-7000 SoC Partition Header
- Zynq-7000 SoC Image Header
- Zynq-7000 SoC Authentication Certificate

Additionally, the Boot Header contains a Zynq-7000 SoC Register Initialization Table. BootROM uses the Boot Header to find the location and length of FSBL and other details to initialize the system before handing off the control to FSBL.

The following table provides the address offsets, parameters, and descriptions for the Zynq®-7000 SoC Boot Header.

### Table 1: Zynq-7000 SoC Boot Header

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x1F</td>
<td>Arm® Vector table</td>
<td>Filled with dummy vector table by Bootgen (Arm Op code 0xEAFFFFFE, which is a branch-to-self infinite loop intended to catch uninitialized vectors.</td>
</tr>
<tr>
<td>0x20</td>
<td>Width Detection Word</td>
<td>This is required to identify the QSPI flash in single/dual stacked or dual parallel mode. 0xAA995566 in little endian format.</td>
</tr>
<tr>
<td>0x24</td>
<td>Header Signature</td>
<td>Contains 4 bytes ‘X’, ‘N’, ‘L’, ‘X’ in byte order, which is 0x584c4e58 in little endian format.</td>
</tr>
<tr>
<td>0x28</td>
<td>Key Source</td>
<td>Location of encryption key within the device: 0x3A5C3C5A: Encryption key in BBRAM. 0xA5C3C3A3: Encryption key in eFUSE. 0x00000000: Not Encrypted.</td>
</tr>
<tr>
<td>0x2C</td>
<td>Header Version</td>
<td>0x01010000</td>
</tr>
<tr>
<td>0x30</td>
<td>Source Offset</td>
<td>Location of FSBL (bootloader) in this image file.</td>
</tr>
<tr>
<td>0x34</td>
<td>FSBL Image Length</td>
<td>Length of the FSBL, after decryption.</td>
</tr>
<tr>
<td>0x38</td>
<td>FSBL Load Address (RAM)</td>
<td>Destination RAM address to which to copy the FSBL.</td>
</tr>
<tr>
<td>0x3C</td>
<td>FSBL Execution address (RAM)</td>
<td>Entry vector for FSBL execution.</td>
</tr>
<tr>
<td>0x40</td>
<td>Total FSBL Length</td>
<td>Total size of FSBL after encryption, including authentication certificate (if any) and padding.</td>
</tr>
<tr>
<td>0x44</td>
<td>QSPI Configuration Word</td>
<td>Hard coded to 0x00000001.</td>
</tr>
</tbody>
</table>
Table 1: Zynq-7000 SoC Boot Header (cont’d)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x48</td>
<td>Boot Header Checksum</td>
<td>Sum of words from offset 0x20 to 0x44 inclusive. The words are assumed to be little endian.</td>
</tr>
<tr>
<td>0x4c-0x97</td>
<td>User Defined Fields</td>
<td>76 bytes</td>
</tr>
<tr>
<td>0x98</td>
<td>Image Header Table Offset</td>
<td>Pointer to Image Header Table (word offset).</td>
</tr>
<tr>
<td>0x9C</td>
<td>Partition Header Table Offset</td>
<td>Pointer to Partition Header Table (word offset).</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Register Initialization Table

The Register Initialization Table in Bootgen is a structure of 256 address-value pairs used to initialize PS registers for MIO multiplexer and flash clocks. For more information, see About Register Initialization Pairs and INT File Attributes.

Table 2: Zynq-7000 SoC Register Initialization Table

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA0 to 0x89C</td>
<td>Register Initialization Pairs: &lt;address&gt;:&lt;value&gt;:</td>
<td>Address = 0xFFFFFFFF means skip that register and ignore the value. All the unused register fields must be set to Address=0xFFFFFFFF and value = 0x0.</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Image Header Table

Bootgen creates a boot image by extracting data from ELF files, bitstream, data files, and so forth. These files, from which the data is extracted, are referred to as images. Each image can have one or more partitions. The Image Header table is a structure, containing information which is common across all these images, and information like; the number of images, partitions present in the boot image, and the pointer to the other header tables. The following table provides the address offsets, parameters, and descriptions for the Zynq®-7000 SoC device.

Table 3: Zynq-7000 SoC Image Header Table

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Version</td>
<td>0x01010000: Only fields available are 0x0, 0x4, 0x8, 0xC, and a padding 0x01020000:0x10 field is added.</td>
</tr>
<tr>
<td>0x04</td>
<td>Count of Image Headers</td>
<td>Indicates the number of image headers.</td>
</tr>
<tr>
<td>0x08</td>
<td>First Partition Header Offset</td>
<td>Pointer to first partition header. (word offset)</td>
</tr>
<tr>
<td>0x0C</td>
<td>First Image Header Offset</td>
<td>Pointer to first image header. (word offset)</td>
</tr>
<tr>
<td>0x10</td>
<td>Header Authentication Certificate Offset</td>
<td>Pointer to the authentication certificate header. (word offset)</td>
</tr>
</tbody>
</table>
Table 3: Zynq-7000 SoC Image Header Table (cont’d)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x14</td>
<td>Reserved</td>
<td>Defaults to 0xFFFFFFFF.</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Image Header

The Image Header is an array of structures containing information related to each image, such as an ELF file, bitstream, data files, and so forth. Each image can have multiple partitions, for example an ELF can have multiple loadable sections, each of which forms a partition in the boot image. The table will also contain the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the Zynq®-7000 SoC device.

Table 4: Zynq-7000 SoC Image Header

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Next Image Header.</td>
<td>Link to next Image Header. 0 if last Image Header (word offset).</td>
</tr>
<tr>
<td>0x04</td>
<td>Corresponding partition header.</td>
<td>Link to first associated Partition Header (word offset).</td>
</tr>
<tr>
<td>0x08</td>
<td>Reserved</td>
<td>Always 0.</td>
</tr>
<tr>
<td>0x0C</td>
<td>Partition Count Length</td>
<td>Number of partitions associated with this image.</td>
</tr>
<tr>
<td>0x10 to N</td>
<td>Image Name</td>
<td>Packed in big-endian order. To reconstruct the string, unpack 4 bytes at a time, reverse the order, and concatenate. For example, the string &quot;FSBL10.ELF&quot; is packed as 0x10: ‘L’, ‘B’, ‘E’, ‘F’, 0x14: ‘E’, ‘0’, ‘1’, 0x18: ‘0’, ‘1’, ‘0’; 0x20: ‘0’, ‘1’, ‘0’, ‘F’. The packed image name is a multiple of 4 bytes.</td>
</tr>
<tr>
<td>N</td>
<td>String Terminator</td>
<td>0x0000000000</td>
</tr>
<tr>
<td>N+4</td>
<td>Reserved</td>
<td>Defaults to 0xFFFFFFFF to 64 bytes boundary.</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Partition Header

The Partition Header is an array of structures containing information related to each partition. Each partition header table is parsed by the Boot Loader. The information such as the partition size, address in flash, load address in RAM, encrypted/signed, and so forth, are part of this table. There is one such structure for each partition including FSBL. The last structure in the table is marked by all NULL values (except the checksum.) The following table shows the offsets, names, and notes regarding the Zynq®-7000 SoC Partition Header.
**Note:** An ELF file with three (3) loadable sections has one image header and three (3) partition header tables.

### Table 5: Zynq-7000 SoC Partition Header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Encrypted Partition length</td>
<td>Encrypted partition data length.</td>
</tr>
<tr>
<td>0x04</td>
<td>Unencrypted Partition length</td>
<td>Unencrypted data length.</td>
</tr>
<tr>
<td>0x08</td>
<td>Total partition word length (Includes Authentication Certificate)</td>
<td>The total partition word length comprises the encrypted information length with padding, the expansion length, and the authentication length.</td>
</tr>
<tr>
<td>0x0C</td>
<td>Destination load address.</td>
<td>The RAM address into which this partition is to be loaded.</td>
</tr>
<tr>
<td>0x10</td>
<td>Destination execution address.</td>
<td>Entry point of this partition when executed.</td>
</tr>
<tr>
<td>0x14</td>
<td>Data word offset in Image</td>
<td>Position of the partition data relative to the start of the boot image</td>
</tr>
<tr>
<td>0x18</td>
<td>Attribute Bits</td>
<td>See Zynq-7000 SoC Partition Attribute Bits</td>
</tr>
<tr>
<td>0x1C</td>
<td>Section Count</td>
<td>Number of sections in a single partition.</td>
</tr>
<tr>
<td>0x20</td>
<td>Checksum Word Offset</td>
<td>Location of the corresponding checksum word in the boot image.</td>
</tr>
<tr>
<td>0x24</td>
<td>Image Header Word Offset</td>
<td>Location of the corresponding Image Header in the boot image.</td>
</tr>
<tr>
<td>0x28</td>
<td>Authentication Certification Word Offset</td>
<td>Location of the corresponding Authentication Certification in the boot image.</td>
</tr>
<tr>
<td>0x2C-0x38</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3C</td>
<td>Header Checksum</td>
<td>Sum of the previous words in the Partition Header.</td>
</tr>
</tbody>
</table>

### Zynq-7000 SoC Partition Attribute Bits

The following table describes the Partition Attribute bits of the partition header table for a Zynq®-7000 SoC device.

### Table 6: Zynq-7000 SoC Partition Attribute Bits

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:18</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
<tr>
<td>17:16</td>
<td>Partition owner</td>
<td>0: FSBL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: UBOOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 and 3: reserved</td>
</tr>
<tr>
<td>15</td>
<td>RSA signature present</td>
<td>0: No RSA authentication certificate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: RSA authentication certificate</td>
</tr>
</tbody>
</table>
Table 6: Zynq-7000 SoC Partition Attribute Bits (cont'd)

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:12</td>
<td>Checksum type</td>
<td>0: None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: MD5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-7: reserved</td>
</tr>
<tr>
<td>11:8</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
<tr>
<td>7:4</td>
<td>Destination device</td>
<td>0: None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: PS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: PL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: INT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-15: Reserved</td>
</tr>
<tr>
<td>3:2</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
<tr>
<td>1:0</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Authentication Certificate

The Authentication Certificate is a structure that contains all the information related to the authentication of a partition. This structure has the public keys, all the signatures that BootROM/FSBL needs to verify. There is an Authentication Header in each Authentication Certificate, which gives information like the key sizes, algorithm used for signing, and so forth. The Authentication Certificate is appended to the actual partition, for which authentication is enabled. If authentication is enabled for any of the partitions, the header tables also need authentication. Header Table Authentication Certificate is appended at end of the header tables content.

The Zynq®-7000 SoC uses an RSA-2048 authentication with a SHA-256 hashing algorithm, which means the primary and secondary key sizes are 2048-bit. Because SHA-256 is used as the secure hash algorithm, the FSBL, partition, and authentication certificates must be padded to a 512-bit boundary.

The format of the Authentication Certificate in a Zynq®-7000 SoC is as shown in the following table.

Table 7: Zynq-7000 SoC Authentication Certificate

<table>
<thead>
<tr>
<th>Authentication Certificate Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Authentication Header = 0x0101000. See Zynq-7000 SoC Authentication Certificate Header.</td>
</tr>
<tr>
<td>0x04</td>
<td>Certificate size</td>
</tr>
<tr>
<td>0x08</td>
<td>UDF (56 bytes)</td>
</tr>
</tbody>
</table>
Table 7: Zynq-7000 SoC Authentication Certificate (cont’d)

<table>
<thead>
<tr>
<th>Authentication Certificate Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>PPK Mod (256 bytes)</td>
</tr>
<tr>
<td>0x140</td>
<td>Mod Ext (256 bytes)</td>
</tr>
<tr>
<td>0x240</td>
<td>Exponent</td>
</tr>
<tr>
<td>0x244</td>
<td>Pad (60 bytes)</td>
</tr>
<tr>
<td>0x280</td>
<td>SPK Mod (256 bytes)</td>
</tr>
<tr>
<td>0x380</td>
<td>Mod Ext (256 bytes)</td>
</tr>
<tr>
<td>0x480</td>
<td>Exponent (4 bytes)</td>
</tr>
<tr>
<td>0x484</td>
<td>Pad (60 bytes)</td>
</tr>
<tr>
<td>0x4C0</td>
<td>SPK Signature = RSA-2048 (PSK, Padding</td>
</tr>
<tr>
<td>0x5C0</td>
<td>FSBL Partition Signature = RSA-2048 (SSK, SHA-256 (Boot Header</td>
</tr>
<tr>
<td>0x5C0</td>
<td>Other Partition Signature = RSA-2048 (SSK, SHA-256 (Partition</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Authentication Certificate Header

The following table describes the Zynq®-7000 SoC Authentication Certificate Header.

Table 8: Zynq-7000 SoC Authentication Certificate Header

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>Reserved</td>
<td>0</td>
</tr>
<tr>
<td>15:14</td>
<td>Authentication Certificate Format</td>
<td>00: PKCS #1 v1.5</td>
</tr>
<tr>
<td>13:12</td>
<td>Authentication Certificate Version</td>
<td>00: Current AC</td>
</tr>
<tr>
<td>11</td>
<td>PPK Key Type</td>
<td>0: Hash Key</td>
</tr>
<tr>
<td>10:9</td>
<td>PPK Key Source</td>
<td>0: eFUSE</td>
</tr>
<tr>
<td>8</td>
<td>SPK Enable</td>
<td>1: SPK Enable</td>
</tr>
<tr>
<td>7:4</td>
<td>Public Strength</td>
<td>0:2048</td>
</tr>
<tr>
<td>3:2</td>
<td>Hash Algorithm</td>
<td>0: SHA256</td>
</tr>
</tbody>
</table>

Zynq-7000 SoC Boot Image Block Diagram

The following is a diagram of the components that can be included in a Zynq®-7000 SoC boot image.
Figure 2: Zynq-7000 SoC Boot Image Block Diagram
Zynq UltraScale+ MPSoC Boot and Configuration

Introduction

Zynq® UltraScale+™ MPSoC supports the ability to boot from different devices such as a QSPI flash, an SD card, USB device firmware upgrade (DFU) host, and the NAND flash drive. This chapter details the boot-up process using different booting devices in both secure and non-secure modes. The boot-up process is managed and carried out by the Platform Management Unit (PMU) and Configuration Security Unit (CSU).

During initial boot, the following steps occur:

- The PMU is brought out of reset by the power on reset (POR).
- The PMU executes code from PMU ROM.
- The PMU initializes the SYSMON and required PLLs for the boot, clears the low power and full power domains, and releases the CSU reset.

After the PMU releases the CSU, CSU does the following:

- Checks to determine if authentication is required by the FSBL or the user application.
- Performs an authentication check and proceeds only if the authentication check passes. Then checks the image for any encrypted partitions.
- If the CSU detects partitions that are encrypted, the CSU performs decryption and initializes OCM, determines boot mode settings, performs the FSBL load and an optional PMU firmware load.
- After execution of CSU ROM code, it hands off control to FSBL. FSBL uses PCAP interface to program the PL with bitstream.

FSBL then takes the responsibility of the system. The Zynq UltraScale+ Device Technical Reference Manual (UG1085) provides details on CSU and PMU. For specific information on CSU, see this link to the "Configuration Security Unit" section of the Zynq UltraScale+ MPSoC: Software Developers Guide (UG1137).

Zynq UltraScale+ MPSoC Boot Image

The following figure shows the Zynq® UltraScale+™ MPSoC boot image.
Figure 3: Zynq UltraScale+ MPSoC Boot Image

<table>
<thead>
<tr>
<th>Boot Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Initialization Table</td>
</tr>
<tr>
<td>PUF Helper Data (Optional)</td>
</tr>
<tr>
<td>Image Header Table</td>
</tr>
<tr>
<td>Image Header 1</td>
</tr>
<tr>
<td>Partition Header 1</td>
</tr>
<tr>
<td>Header Authentication Certificate (Optional)</td>
</tr>
<tr>
<td>Partition 1 (FSBL)</td>
</tr>
<tr>
<td>Partition 2</td>
</tr>
<tr>
<td>Partition n</td>
</tr>
</tbody>
</table>
Zynq UltraScale+ MPSoC Boot Header

About the Boot Header

Bootgen attaches a boot header at the starting of any boot image. The Boot Header table is a structure that contains information related to booting of primary bootloader, such as the FSBL. There is only one such structure in entire boot image. This table is parsed by BootROM to get the information of where FSBL is stored in flash and where it needs to be loaded in OCM. Some encryption and authentication related parameters are also stored in here. The Boot image components are:

- Zynq UltraScale+ MPSoC Boot Header, which also has the Zynq UltraScale+ MPSoC Boot Header Attribute Bits.
- Zynq UltraScale+ MPSoC Register Initialization Table
- Zynq UltraScale+ MPSoC PUF Helper Data
- Zynq UltraScale+ MPSoC Image Header Table
- Zynq UltraScale+ MPSoC Image Header
- Zynq UltraScale+ MPSoC Authentication Certificates
- Zynq UltraScale+ MPSoC Partition Header

BootROM uses the Boot Header to find the location and length of FSBL and other details to initialize the system before handing off the control to FSBL. The following table provides the address offsets, parameters, and descriptions for the Zynq® UltraScale+™ MPSoC device.

Table 9: Zynq UltraScale+ MPSoC Device Boot Header

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x1F</td>
<td>Arm® vector table</td>
<td>XIP ELF vector table: 0xEAFFFFFE: for Cortex™-R5F and Cortex A53 (32-bit) 0x14000000: for Cortex A53 (64-bit)</td>
</tr>
<tr>
<td>0x20</td>
<td>Width Detection Word</td>
<td>This field is used for QSPI width detection. 0xAA995566 in little endian format.</td>
</tr>
<tr>
<td>0x24</td>
<td>Header Signature</td>
<td>Contains 4 bytes ‘X’, ‘N’, ‘L’, ‘X’ in byte order, which is 0x584c4e58 in little endian format.</td>
</tr>
<tr>
<td>0x28</td>
<td>Key Source</td>
<td>0x00000000 (Un-Encrypted) 0xA5C3C5A5 (Black key stored in eFUSE) 0xA5C3C5A7 (Obfuscated key stored in eFUSE) 0x3A5C3C5A (Red key stored in BBRAM) 0xA5C3C5A3 (eFUSE RED key stored in eFUSE) 0xA35C7CA5 (Obfuscated key stored in Boot Header) 0xA3A5C3C5 (USER key stored in Boot Header) 0xA35C7C53 (Black key stored in Boot Header)</td>
</tr>
</tbody>
</table>
### Table 9: Zynq UltraScale+ MPSoC Device Boot Header (cont’d)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2C</td>
<td>FSBL Execution address (RAM)</td>
<td>FSBL execution address in OCM or XIP base address.</td>
</tr>
<tr>
<td>0x30</td>
<td>Source Offset</td>
<td>If no PMUFW, then it is the start offset of FSBL. If PMUFW, then start of PMUFW.</td>
</tr>
<tr>
<td>0x34</td>
<td>PMU Image Length</td>
<td>PMU FW original image length in bytes. (0-128KB).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If size &gt; 0, PMUFW is prefixed to FSBL. If size = 0, no PMUFW image.</td>
</tr>
<tr>
<td>0x38</td>
<td>Total PMU FW Length</td>
<td>Total PMUFW image length in bytes. (PMUFW length + encryption overhead)</td>
</tr>
<tr>
<td>0x3C</td>
<td>FSBL Image Length</td>
<td>Original FSBL image length in bytes. (0-250KB). If 0, XIP bootimage is assumed.</td>
</tr>
<tr>
<td>0x40</td>
<td>Total FSBL Length</td>
<td>FSBL image length + Encryption overhead of FSBL image + Auth. Cert., + 64byte alignment + hash size (Integrity check).</td>
</tr>
<tr>
<td>0x44</td>
<td>FSBL Image Attributes</td>
<td>See Bit Attributes.</td>
</tr>
<tr>
<td>0x48</td>
<td>Boot Header Checksum</td>
<td>Sum of words from offset 0x20 to 0x44 inclusive. The words are assumed to be little endian.</td>
</tr>
<tr>
<td>0x4C-0x68</td>
<td>Obfuscated/Black Key Storage</td>
<td>Stores the Obfuscated key or Black key.</td>
</tr>
<tr>
<td>0x6C</td>
<td>Shutter Value</td>
<td>32-bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time.</td>
</tr>
<tr>
<td>0x70-0x94</td>
<td>User-Defined Fields (UDF)</td>
<td>40 bytes.</td>
</tr>
<tr>
<td>0x98</td>
<td>Image Header Table Offset</td>
<td>Pointer to Image Header Table. (word offset)</td>
</tr>
<tr>
<td>0x9C</td>
<td>Partition Header Table Offset</td>
<td>Pointer to Partition Header. (word offset)</td>
</tr>
<tr>
<td>0xA0-0xA8</td>
<td>Secure Header IV</td>
<td>IV for secure header of bootloader partition.</td>
</tr>
<tr>
<td>0x0AC-0xB4</td>
<td>Obfuscated/Black Key IV</td>
<td>IV for Obfuscated or Black key.</td>
</tr>
</tbody>
</table>

### Zynq UltraScale+ MPSoC Boot Header Attribute Bits

#### Table 10: Zynq UltraScale+ MPSoC Boot Header Attribute Bits

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Offset</th>
<th>Width</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>31:16</td>
<td>16</td>
<td>0x0</td>
<td>Reserved. Must be 0.</td>
</tr>
<tr>
<td>BHDR RSA</td>
<td>15:14</td>
<td>2</td>
<td>0x0</td>
<td>0x3: RSA Authentication of the boot image will be done, excluding verification of PPK hash and SPK ID. All Others others: RSA Authentication will be decided based on eFuse RSA bits.</td>
</tr>
</tbody>
</table>
Table 10: Zynq UltraScale+ MPSoC Boot Header Attribute Bits (cont’d)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bit Offset</th>
<th>Width</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>13:12</td>
<td>2</td>
<td>0x0</td>
<td>NA</td>
</tr>
<tr>
<td>CPU Select</td>
<td>11:10</td>
<td>2</td>
<td>0x0</td>
<td>0x0: R5 Single 0x1: A53 Single 0x2: R5 Dual 0x3: Reserved</td>
</tr>
<tr>
<td>Hashing Select</td>
<td>9:8</td>
<td>2</td>
<td>0x0</td>
<td>0x0, 0x1 : No Integrity check 0x3: SHA3 for BI integrity check</td>
</tr>
<tr>
<td>PUF-HD</td>
<td>7:6</td>
<td>2</td>
<td>0x0</td>
<td>0x3: PUF HD is part of boot header. All other: PUF HD is in eFuse</td>
</tr>
<tr>
<td>Reserved</td>
<td>5:0</td>
<td>6</td>
<td>0x0</td>
<td>Reserved for future use. Must be 0.</td>
</tr>
</tbody>
</table>

Zynq UltraScale+ MPSoC Register Initialization Table

The Register Initialization Table in Bootgen is a structure of 256 address-value pairs used to initialize PS registers for MIO multiplexer and flash clocks. For more information, see About Register Initialization Pairs and INT File Attributes.

Table 11: Zynq UltraScale+ MPSoC Register Initialization Table

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xB8 to 0x8B4</td>
<td>Register Initialization Pairs: &lt;address&gt;:&lt;value&gt;: (2048 bytes)</td>
<td>If the Address is set to 0xFFFFFFFF, that register is skipped and the value is ignored. All unused register fields must be set to Address=0xFFFFFFFF and value =0x0.</td>
</tr>
</tbody>
</table>

Zynq UltraScale+ MPSoC PUF Helper Data

The PUF uses helper data to re-create the original KEK value over the complete guaranteed operating temperature and voltage range over the life of the part. The helper data consists of a <syndrome_value>, an <aux_value>, and a <chash_value>. The helper data can either be stored in eFUSEs or in the boot image. See puf_file for more information. Also, see this link to the section on "PUF Helper Data" in Zynq UltraScale+ Device Technical Reference Manual (UG1085).
### Table 12: Zynq UltraScale+ MPSoC PUF Helper Data

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8B8 to 0xEC0</td>
<td>PUF Helper Data (1544 bytes)</td>
<td>Valid only when Boot Header Offset 0x44 (bits 7:6) == 0x3. If the PUF HD is not inserted then Boot Header size = 2048 bytes. If the PUF Header Data is inserted, then the Boot Header size = 3584 bytes. PUF HD size = Total size = 1536 bytes of PUFHD + 4 bytes of CHASH + 2 bytes of AUX + 1 byte alignment = 1544 byte.</td>
</tr>
</tbody>
</table>

### Zynq UltraScale+ MPSoC Image Header Table

Bootgen creates a boot image by extracting data from ELF files, bitstream, data files, and so forth. These files, from which the data is extracted, are referred to as images. Each image can have one or more partitions. The Image Header table is a structure, containing information which is common across all these images, and information like; the number of images, partitions present in the boot image, and the pointer to the other header tables.

### Table 13: Zynq UltraScale+ MPSoC Device Image Header Table

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Version</td>
<td>0x01010000 0x01020000 - 0x10 field is added</td>
</tr>
<tr>
<td>0x04</td>
<td>Count of Image Header</td>
<td>Indicates the number of image headers.</td>
</tr>
<tr>
<td>0x08</td>
<td>1st Partition Header Offset</td>
<td>Pointer to first partition header. (word offset)</td>
</tr>
<tr>
<td>0x0C</td>
<td>1st Image Offset Header</td>
<td>Pointer to first image header. (word offset)</td>
</tr>
<tr>
<td>0x10</td>
<td>Header Authentication Certificate</td>
<td>Options are: 0 - Same boot device 1 - QSPI-32 2 - QSPI-24 3 - NAND 4 - SD0 4 - SD1 5 - SDLS 6 - MMC 7 - USB 8 - ETHERNET 9 - PCIE 10 – SATA</td>
</tr>
<tr>
<td>0x14</td>
<td>Secondary Boot Device</td>
<td>Options are: 0 - Same boot device 1 - QSPI-32 2 - QSPI-24 3 - NAND 4 - SD0 4 - SD1 5 - SDLS 6 - MMC 7 - USB 8 - ETHERNET 9 - PCIE 10 – SATA</td>
</tr>
<tr>
<td>0x18- 0x38</td>
<td>Padding</td>
<td>Reserved (0x0)</td>
</tr>
<tr>
<td>0x3C</td>
<td>Checksum</td>
<td>A sum of all the previous words in the image header.</td>
</tr>
</tbody>
</table>
Zynq UltraScale+ MPSoC Image Header

About Image Headers

The Image Header is an array of structures containing information related to each image, such as an ELF file, bitstream, data files, and so forth. Each image can have multiple partitions, for example an ELF can have multiple loadable sections, each of which form a partition in the boot image. The table will also contain the information of number of partitions related to an image.

The following table provides the address offsets, parameters, and descriptions for the Zynq® UltraScale+™ MPSoC.

Table 14: Zynq UltraScale+ MPSoC Device Image Header

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Next image header offset</td>
<td>Link to next Image Header. 0 if last Image Header. (word offset)</td>
</tr>
<tr>
<td>0x04</td>
<td>Corresponding partition header</td>
<td>Link to first associated Partition Header. (word offset)</td>
</tr>
<tr>
<td>0x08</td>
<td>Reserved</td>
<td>Always 0.</td>
</tr>
<tr>
<td>0x0C</td>
<td>Partition Count</td>
<td>Value of the actual partition count.</td>
</tr>
<tr>
<td>0x10 - N</td>
<td>Image Name</td>
<td>Packed in big-endian order. To reconstruct the string, unpack 4 bytes at a time, reverse the order, and concatenated. For example, the string &quot;FSBL10.ELF&quot; is packed as 0x10: 'L','B','S','F'. 0x14: 'E','.','0','1'. 0x18: '\0','F','L'. The packed image name is a multiple of 4 bytes.</td>
</tr>
<tr>
<td>varies</td>
<td>String Terminator</td>
<td>0x000000</td>
</tr>
<tr>
<td>varies</td>
<td>Padding</td>
<td>Defaults to 0xFFFFFFF to 64 bytes boundary.</td>
</tr>
</tbody>
</table>

Zynq UltraScale+ MPSoC Partition Header

About the Partition Header

The Partition Header is an array of structures containing information related to each partition. Each partition header table is parsed by the Boot Loader. The information such as the partition size, address in flash, load address in RAM, encrypted/signed, and so forth, are part of this table. There is one such structure for each partition including FSBL. The last structure in the table is marked by all NULL values (except the checksum.) The following table shows the offsets, names, and notes regarding the Zynq® UltraScale+™ MPSoC.

Table 15: Zynq UltraScale+ MPSoC Device Partition Header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Encrypted Partition Data Word Length</td>
<td>Encrypted partition data length.</td>
</tr>
<tr>
<td>0x04</td>
<td>Un-encrypted Data Word Length</td>
<td>Unencrypted data length.</td>
</tr>
</tbody>
</table>
Table 15: Zynq UltraScale+ MPSoC Device Partition Header (cont’d)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>Total Partition Word Length</td>
<td>The total encrypted + padding + expansion +authentication length.</td>
</tr>
<tr>
<td></td>
<td>(Includes Authentication Certificate. See Authentication Certificate.)</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td>Next Partition Header Offset LO</td>
<td>Location of next partition header (word offset).</td>
</tr>
<tr>
<td>0x10</td>
<td>Destination Execution Address</td>
<td>The lower 32-bits of executable address of this partition after loading.</td>
</tr>
<tr>
<td>0x14</td>
<td>Destination Execution Address HI</td>
<td>The higher 32-bits of executable address of this partition after loading.</td>
</tr>
<tr>
<td>0x18</td>
<td>Destination Load Address LO</td>
<td>The lower 32-bits of RAM address into which this partition is to be loaded.</td>
</tr>
<tr>
<td>0x1C</td>
<td>Destination Load Address HI</td>
<td>The higher 32-bits of RAM address into which this partition is to be loaded.</td>
</tr>
<tr>
<td>0x20</td>
<td>Actual Partition Word Offset</td>
<td>The position of the partition data relative to the start of the boot image. (word offset)</td>
</tr>
<tr>
<td>0x24</td>
<td>Attributes</td>
<td>See Zynq UltraScale+ MPSoC Partition Attribute Bits</td>
</tr>
<tr>
<td>0x28</td>
<td>Section Count</td>
<td>The number of sections associated with this partition.</td>
</tr>
<tr>
<td>0x2C</td>
<td>Checksum Word Offset</td>
<td>The location of the checksum table in the boot image. (word offset)</td>
</tr>
<tr>
<td>0x30</td>
<td>Image Header Word Offset</td>
<td>The location of the corresponding image header in the boot image. (word offset)</td>
</tr>
<tr>
<td>0x34</td>
<td>Partition Number/ID</td>
<td>Partition ID.</td>
</tr>
<tr>
<td>0x3C</td>
<td>Header Checksum</td>
<td>A sum of the previous words in the Partition Header.</td>
</tr>
</tbody>
</table>

Zynq UltraScale+ MPSoC Partition Attribute Bits

The following table describes the Partition Attribute bits on the partition header table for the Zynq® UltraScale+™ MPSoC.

Table 16: Zynq® UltraScale+™ MPSoC Device Partition Attribute Bits

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Vector Location</td>
<td>Location of exception vector. 0: LOVEC (default) 1: HIVEC</td>
</tr>
<tr>
<td>22:20</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Early Handoff</td>
<td>Handoff immediately after loading: 0: No Early Handoff 1: Early Handoff Enabled</td>
</tr>
<tr>
<td>18</td>
<td>Endianness</td>
<td>0: Little Endian 1: Big Endian</td>
</tr>
</tbody>
</table>
### Table 16: Zynq® UltraScale™ MPSoC Device Partition Attribute Bits (cont’d)

<table>
<thead>
<tr>
<th>Bit Offset</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 17:16      | Partition Owner | 0: FSBL  
1: U-Boot  
2 and 3: Reserved |
| 15         | RSA Authentication Certificate present | 0: No RSA Authentication Certificate  
1: RSA Authentication Certificate |
| 14:12      | Checksum Type | 0: None  
1-2: Reserved  
3: SHA3  
4-7: Reserved |
| 11:8       | Destination CPU | 0: None  
1: A53-0  
2: A53-1  
3: A53-2  
4: A53-3  
5: R5-0  
6: R5-1  
7 R5-lockstep  
8: PMU  
9-15: Reserved |
| 7          | Encryption Present | 0: Not Encrypted  
1: Encrypted |
| 6:4        | Destination Device | 0: None  
1: PS  
2: PL  
3-15: Reserved |
| 3          | ASX Exec State | 0: AARCH64 (default)  
1: AARCH32 |
| 2:1        | Exception Level | 0: EL0  
1: EL1  
2: EL2  
3: EL3 |
| 0          | Trustzone | 0: Non-secure  
1: Secure |
Zynq UltraScale+ MPSoC Authentication Certificates

The Authentication Certificate is a structure that contains all the information related to the authentication of a partition. This structure has the public keys and the signatures that BootROM/FSBL needs to verify. There is an Authentication Header in each Authentication Certificate, which gives information like the key sizes, algorithm used for signing, and so forth. The Authentication Certificate is appended to the actual partition, for which authentication is enabled. If authentication is enabled for any of the partitions, the header tables also need authentication. The Header Table Authentication Certificate is appended at end of the content to the header tables.

The Zynq® UltraScale+™ MPSoC uses RSA-4096 authentication, which means the primary and secondary key sizes are 4096-bit. The following table provides the format of the Authentication Certificate for the Zynq UltraScale+ MPSoC device.

Table 17: Zynq UltraScale+ MPSoC Device Authentication Certificates

<table>
<thead>
<tr>
<th>Authentication Certificate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Authentication Header = 0x0101000. See Zynq UltraScale+ MPSoC Authentication Certification Header.</td>
</tr>
<tr>
<td>0x04</td>
<td>SPK ID</td>
</tr>
<tr>
<td>0x08</td>
<td>UDF (56 bytes)</td>
</tr>
<tr>
<td>0x40</td>
<td>PPK</td>
</tr>
<tr>
<td>0x240</td>
<td></td>
</tr>
<tr>
<td>0x440</td>
<td></td>
</tr>
<tr>
<td>0x444</td>
<td></td>
</tr>
<tr>
<td>0x480</td>
<td>SPK</td>
</tr>
<tr>
<td>0x680</td>
<td></td>
</tr>
<tr>
<td>0x880</td>
<td></td>
</tr>
<tr>
<td>0x884</td>
<td></td>
</tr>
<tr>
<td>0x8C0</td>
<td>SPK Signature = RSA-4096 (PSK, Padding</td>
</tr>
<tr>
<td>0xAC0</td>
<td>Boot Header Signature = RSA-4096 (SSK, Padding</td>
</tr>
<tr>
<td>0xCC0</td>
<td>Partition Signature = RSA-4096 (SSK, Padding</td>
</tr>
</tbody>
</table>

Note: FSBL Signature is calculated as follows:

FSBL Signature = RSA-4096 (SSK, Padding || SHA-384 (PMUFW || FSBL || Padding || Authentication Header || UDF || PPK || SPK || SPK Signature))

Zynq UltraScale+ MPSoC Authentication Certification Header

The following table describes the Authentication Header bit fields for the Zynq® UltraScale+™ MPSoC device.

Chapter 2: Boot Image Layout
<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:20</td>
<td>Reserved</td>
<td>0</td>
</tr>
<tr>
<td>19:18</td>
<td>SPK/User eFuse Select</td>
<td>01: SPK eFuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10: User eFuse</td>
</tr>
<tr>
<td>17:16</td>
<td>PPK Key Select</td>
<td>0: PPK0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: PPK1</td>
</tr>
<tr>
<td>15:14</td>
<td>Authentication Certificate</td>
<td>00: PKCS #1 v1.1</td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>13:12</td>
<td>Authentication Certificate</td>
<td>00: Current AC</td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>PPK Key Type</td>
<td>0: Hash Key</td>
</tr>
<tr>
<td>10:9</td>
<td>PPK Key Source</td>
<td>0: eFUSE</td>
</tr>
<tr>
<td>8</td>
<td>SPK Enable</td>
<td>1: SPK Enable</td>
</tr>
<tr>
<td>7:4</td>
<td>Public Strength</td>
<td>0 : 2048b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 : 4096</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2:3 : Reserved</td>
</tr>
<tr>
<td>3:2</td>
<td>Hash Algorithm</td>
<td>1: SHA3/384</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2:3 Reserved</td>
</tr>
<tr>
<td>1:0</td>
<td>Public Algorithm</td>
<td>0: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: RSA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: Reserved</td>
</tr>
</tbody>
</table>

**Zynq UltraScale+ MPSoC Secure Header**

When you choose to encrypt a partition, Bootgen appends the secure header to that partition. The secure header, contains the key/iv used to encrypt the actual partition. This header in-turn is encrypted using the device key and iv. The Zynq UltraScale+ Secure header is shown in the following table.
### Zynq UltraScale+ MPSoC Boot Image Block Diagram

The following is a diagram of the components that can be included in a Zynq® UltraScale+™ MPSoC boot image.

#### AES

<table>
<thead>
<tr>
<th></th>
<th>Partition#0 (FSBL)</th>
<th>Partition#1</th>
<th>Partition#2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key0</td>
<td>Key0</td>
<td>Key0</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV0</td>
<td>IV0+0x01</td>
<td>IV0+0x02</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key0</td>
<td>Key1</td>
<td>Key0</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV1</td>
<td>IV1</td>
<td>IV1</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key0</td>
<td>Key1</td>
<td>Key0</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV0+0x01</td>
<td>IV0+0x02</td>
<td>IV1</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key1</td>
<td>Key1</td>
<td>Key1</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV1</td>
<td>IV1</td>
<td>IV1</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key0</td>
<td>Key1</td>
<td>Key1</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV0+0x02</td>
<td>Key1</td>
<td>IV1</td>
</tr>
</tbody>
</table>

#### AES with Key rolling

<table>
<thead>
<tr>
<th></th>
<th>Partition#0 (FSBL)</th>
<th>Partition#1</th>
<th>Partition#2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key0</td>
<td>Key2</td>
<td>Key3</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV1</td>
<td>IV2</td>
<td>IV3</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key1</td>
<td>Key2</td>
<td>Key3</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV1</td>
<td>IV2</td>
<td>IV3</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key2</td>
<td>Key3</td>
<td>Key4</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV3</td>
<td>IV4</td>
<td>IV3</td>
</tr>
<tr>
<td><strong>Encrypted Using</strong></td>
<td>Key3</td>
<td>Key4</td>
<td>Key3</td>
</tr>
<tr>
<td><strong>Contents</strong></td>
<td>IV4</td>
<td>IV3</td>
<td>Key4</td>
</tr>
</tbody>
</table>

...
Figure 4: Zynq UltraScale+ MPSoC Device Boot Image Block Diagram
Chapter 3

Creating Boot Images

Boot Image Format (BIF)

The Xilinx® boot image layout has multiple files, file types, and supporting headers to parse those files by boot loaders. Bootgen defines multiple attributes for generating the boot images and interprets and generates the boot images, based on what is passed in the files. Because there are multiple commands and attributes available, Bootgen defines a boot image format (BIF) to contain those inputs. A BIF comprises of the following:

- Configuration attributes to create secure/non-secure boot images
- An FSBL
- One or more Partition Images

Along with properties and attributes, Bootgen takes multiple commands to define the behavior while it is creating the boot images. For example, to create a boot image for a qualified FPGA device, a Zynq®-7000 SoC device, or a Zynq® UltraScale+™ MPSoC device, you should provide the appropriate arch command option to Bootgen. The following appendices list and describe the available options to direct Bootgen behavior.

- Appendix B: BIF Attribute Reference
- Appendix C: Command Reference

The format of the boot image conforms to a hybrid mix of hardware and software requirements. The boot image Header is required by the bootROM loader which loads a single partition, typically the first stage boot loader (FSBL). The remainder of the boot image is loaded and processed by the FSBL. Bootgen generates a boot image by combining a list of partitions. These partitions can be:

- First Stage Boot Loader (FSBL)
- Secondary Stage Boot Loader (SSBL) like U-Boot
- Bitstream
- Linux
- Software applications to run on processors
BIF Syntax and Supported File Types

The BIF file specifies each component of the boot image, in order of boot, and allows optional attributes to be applied to each image component. In some cases, an image component can be mapped to more than one partition if the image component is not contiguous in memory. BIF file syntax takes the following form:

```plaintext
<image_name>:
[
  // common attributes
  [attribute1] <argument1>

  // partition attributes
  [attribute2, attribute3=<argument>] <elf>
  [attribute2, attribute3=<argument>, attribute4=<argument>] <bit>
  [attribute3] <elf>
  <bin>
]
```

- The `<image_name>` and the {...} grouping brackets the files that are to be made into partitions in the ROM image.
- One or more data files are listed in the {...} brackets.
- Supported file types are: ELF, BIT, RBT, INT, or BIN files.
- Each partition data files can have an optional set of attributes preceding the data file name with the syntax `[attribute, attribute=<argument>]`.
- Attributes apply some quality to the data file.
- Multiple attributes can be listed separated with a `[,]` as a separator. The order of multiple attributes is not important. Some attributes are one keyword, some are keyword equates.
- You can also add a filepath to the file name if the file is not in the current directory. How you list the files is free form; either all on one line (separated by any white space, and at least one space), or on separate lines.
- White space is ignored, and can be added for readability.
- You can use C-style block comments of `/* . . . */`, or C++ line comments of `//`. 
The following example is of a BIF with additional white space and new lines for improved readability:

```xml
<image_name>:
{
/* common attributes */
[attribute1] <argument1>

/* bootloader */
[attribute2,
 attribute3,
 attribute4=<argument>
] <elf>

/* pl bitstream */
[
 attribute2,
 attribute3,
 attribute4=<argument>,
 attribute=<argument>
] <bit>

/* another elf partition */
[
 attribute3
] <elf>

/* bin partition */
<bin>
}
```

**Bootgen Supported Files**

The following table lists the Bootgen supported files.

**Table 18: Bootgen Supported Files**

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bin</td>
<td>Binary</td>
<td>Raw binary file.</td>
</tr>
<tr>
<td>.bit/.rbt</td>
<td>Bitstream</td>
<td>Strips the BIT file header.</td>
</tr>
<tr>
<td>.dtb</td>
<td>Binary</td>
<td>Raw binary file.</td>
</tr>
<tr>
<td>image.gz</td>
<td>Binary</td>
<td>Raw binary file.</td>
</tr>
<tr>
<td>.elf</td>
<td>Executable Linked File (ELF)</td>
<td>Symbols and headers removed.</td>
</tr>
<tr>
<td>.int</td>
<td>Register initialization file</td>
<td></td>
</tr>
<tr>
<td>.nky</td>
<td>AES key</td>
<td></td>
</tr>
<tr>
<td>.pki/.pub/.pem</td>
<td>RSA key</td>
<td></td>
</tr>
<tr>
<td>.sig</td>
<td>Signature files</td>
<td>Signature files generated by bootgen or HSM.</td>
</tr>
</tbody>
</table>
## Attributes

The following table lists the Bootgen attributes. Each attribute is linked to a longer description in the left column with a short description in the right column. The architecture name indicates what Xilinx® device uses that attribute:

- zynq: Zynq-7000 SoC device
- zynqmp: Zynq® UltraScale+™ MPSoC device
- fpga: Any 7 series and above devices

### Table 19: Bootgen Attributes and Description

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aeskeyfile</strong> &lt;aes_key_filepath&gt;</td>
<td>The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file needs to written to eFUSE or BRAM. If the key file is not present in the path specified, a new key is generated by bootgen, which is used for encryption. For example: If encryption is selected for bitstream in the BIF file, the output is an encrypted bitstream.</td>
<td>All</td>
</tr>
<tr>
<td><strong>aarch32_mode</strong></td>
<td>To specify the binary file that is to be executed in 32-bit mode.</td>
<td>zynqmp</td>
</tr>
<tr>
<td><strong>alignment</strong> &lt;byte&gt;</td>
<td>Sets the byte alignment. The partition will be padded to be aligned to a multiple of this value. This attribute cannot be used with offset.</td>
<td>zynq, zynqmp</td>
</tr>
</tbody>
</table>
| **auth_params** <options> | Extra options for authentication:  
  - `ppk_select`: 0=1, 1=2 of two PPKs supported.  
  - `spk_id`: 32-bit ID to differentiate SPKs.  
  - `spk_select`: To differentiate spk and user efuses. Default will be spk-efuse.  
  - `header_auth`: To authenticate headers when no partition is authenticated.                                                                                                                                                                                                 | zynqmp                   |
| **authentication** <option> | Specifies the partition to be authenticated.  
  Authentication for Zynq is done using RSA-2048.  
  Authentication for Zynq UltraScale+ MPSoCs is done using RSA-4096.  
  The arguments are:  
  - `none`: Partition not encrypted.  
  - `rsa`: Partition encrypted using RSA algorithm.                                                                                                                                                                                                                           | All                      |
| **bh_keyfile** <filename> | 256-bit obfuscated key or black key to be stored in the Boot Header. This is only valid when `keysrc_encryption` is `bh_gry_key` or `keysrc_encryption` is `bh_blk_key`.                                                                                                                                                                 | zynqmp                   |
| **bh_key_iv** <filename> | Initialization vector used when decrypting the obfuscated key or a black key.                                                                                                                                                                                                                                                           | zynqmp                   |
| **bhsignature** <filename> | Imports Boot Header signature into authentication certificate. This can be used if you do not want to share the secret key PSK. You can create a signature and provide it to Bootgen. The file format is `boothdr_sha384.sig`.                                                                                                                                          | zynqmp                   |
Table 19: Bootgen Attributes and Description (cont’d)

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>big_endian</td>
<td>To specify the binary file is in big endian format</td>
<td>zynqmp</td>
</tr>
<tr>
<td>blocks &lt;block sizes&gt;</td>
<td>Specify block sizes for key-rolling feature in Encryption. Each module is encrypted using its own unique key. The initial key is stored at the key source on the device, while keys for each successive blocks are encrypted (wrapped) in the previous module.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>boot_device &lt;options&gt;</td>
<td>Specifies the secondary boot device. Indicates the device on which the partition is present. Options are: qspi32 qspi24 nand sd0 sd1 sd-ls mmc usb ethernet pcie sata</td>
<td>zynqmp</td>
</tr>
<tr>
<td>bootimage &lt;filename.bin&gt;</td>
<td>Specifies that the listed input file is a boot image that was created by Bootgen.</td>
<td>zynq zynqmp</td>
</tr>
<tr>
<td>bootloader &lt;partition&gt;</td>
<td>Specifies the partition is a bootloader (FSBL). This attribute is specified along with other partition BIF attributes.</td>
<td>zynq zynqmp</td>
</tr>
<tr>
<td>bootvector &lt;vector_values&gt;</td>
<td>Specifies the vector table for execute in place (XIP).</td>
<td>zynqmp</td>
</tr>
<tr>
<td>checksum &lt;options&gt;</td>
<td>Specifies that the partition needs to be checksummed. This option is not supported along with more secure features like authentication and encryption. Checksum algorithms are: md5: for Zynq®-7000 SoC devices. For Zynq® UltraScale™ MPSoC, options are none:No checksum operation. sha3: Sha3 checksum. Zynq devices do not support checksum for boot loaders. Zynq UltraScale+ MPSoC devices do support checksum operation for bootloaders.</td>
<td>zynq zynqmp</td>
</tr>
<tr>
<td>destination_cpu &lt;device_core&gt;</td>
<td>Specifies the core on which the partition needs to be executed.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>Option/Attribute</td>
<td>Description</td>
<td>Used By</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| destination_device <device_type> | This specifies if the partition is targeted for PS or PL. The options are:  
  ps: the partition is targeted for PS (default).  
  pl: the partition is targeted for PL, for bitstreams. | zynqmp   |
| early_handoff              | This flag ensures that the handoff to applications that are critical immediately after the partition is loaded; otherwise, all the partitions are loaded sequentially first, and then the handoff also happens in a sequential fashion. | zynqmp   |
| encryption <option>        | Specifies the partition to be encrypted. Encryption algorithms are: zynq uses AES-CBC, and zynqmp uses AES-GCM.  
  The partition options are:  
  none: Partition not encrypted.  
  aes: Partition encrypted using AES algorithm. | All      |
| exception_level<options>   | Exception level for which the core should be configured.  
  Options are:  
  el-0  
  el-1  
  el-2  
  el-3 | zynqmp   |
| familykey                  | Specifies the family key.                                                   | zynqmp fpga |
| fsbl_config                | Specifies the sub-attributes used to configure the bootimage. Those sub-attributes are:  
  bh_auth_enable: RSA authentication of the boot image is done excluding the verification of PPK hash and SPK ID.  
  auth_only: boot image is only RSA signed. FSBL should not be decrypted.  
  opt_key: Operational key is used for block-0 decryption. Secure Header has the opt key.  
  pufhd_bh: PUF helper data is stored in Boot Header. (Default is efuse).  
  PUF helper data file is passed to Bootgen using the [puf_file] option.  
  puf4kmode: PUF is tuned to use in 4k bit configuration. (Default is 12k bit).  
  shutter = <value>32 bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time. | zynqmp |
| headersignature<signature_file> | Imports the header signature into an Authentication Certificate. This can be used in case the user does not want to share the secret key, The user can create a signature and provide it to Bootgen. | zynq zynqmp |
### Table 19: Bootgen Attributes and Description (cont’d)

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>hivec</td>
<td>Specifies the location of exception vector table as hivec (Hi-Vector). The default value is lovec (Low-Vector). This is applicable with A53 (32 bit) and R5 cores only. hivec: exception vector table at 0xFFFFF0000. lovec: exception vector table at 0x00000000.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>init &lt;filename&gt;</td>
<td>Register initialization block at the end of the Bootloader, built by parsing the init (.int) file specification. A maximum of 256 address-value init pairs are allowed. The init files have a specific format.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>keysrc_encryption</td>
<td>Specifies the Key source for encryption. The keys are: efuse_gry_key: Grey (Obfuscated) Key stored in eFUSE. See Gray/Obfuscated Keys bh_gry_key: Grey (Obfuscated) Key stored in boot header. bh_blk_key: Black Key stored in boot header. See Black/PUF Keys efuse_blk_key: Black Key stored in eFUSE. kup_key: User Key. efuse_red_key: Red key stored in eFUSE. See Rolling Keys bbram_red_key: Red key stored in BBRAM.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>load &lt;partition_address&gt;</td>
<td>Sets the load address for the partition in memory.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>offset &lt;offset_address&gt;</td>
<td>Sets the absolute offset of the partition in the boot image.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>partition_owner &lt;option&gt;</td>
<td>Owner of the partition which is responsible to load the partition. Options are: fsbl: Partition is loaded by FSBL uboot: Partition is loaded by U-Boot.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>pid &lt;ID&gt;</td>
<td>Specifies the Partition ID. PID can be a 32-bit value (0 to 0xFFFFFFFF).</td>
<td>zynqmp</td>
</tr>
<tr>
<td>pmufw_image &lt;image_name&gt;</td>
<td>PMU firmware image to be loaded by BootROM, before loading the FSBL.</td>
<td>zynqmp</td>
</tr>
<tr>
<td>ppkfile &lt;key filename&gt;</td>
<td>Primary Public Key (PPK). Used to authenticate partitions in the boot image. See Using Authentication for more information.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>presign &lt;sig_filename&gt;</td>
<td>Partition signature (.sig) file.</td>
<td>All</td>
</tr>
<tr>
<td>pskfile &lt;key filename&gt;</td>
<td>Primary Secret Key (PSK). Used to authenticate partitions in the boot image. See the Using Authentication for more information.</td>
<td>zynq    zynqmp</td>
</tr>
<tr>
<td>puf_file &lt;filename&gt;</td>
<td>PUF helper data file. PUF is used with black key as encryption key source. PUF helper data is of 1544 bytes.1536 bytes of PUF HD + 4 bytes of HASH + 3 bytes of AUX + 1 byte alignment.</td>
<td>zynqmp</td>
</tr>
</tbody>
</table>
### Table 19: Bootgen Attributes and Description (cont’d)

<table>
<thead>
<tr>
<th>Option/Attribute</th>
<th>Description</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserve</td>
<td>Reserves the memory, which is padded after the partition.</td>
<td>zynq, zynqmp</td>
</tr>
<tr>
<td>spkfile &lt;filename&gt;</td>
<td>Keys used to authenticate partitions in the boot image. See Using Authentication for more information. SPK - Secondary Public Key</td>
<td>All</td>
</tr>
</tbody>
</table>
| split <options> | Splits the image into parts, based on the mode. Split options are:  
  slaveboot: Supported for zynqmp only. Splits as follows:  
  Boot Header + Bootloader  
  Image and Partition Headers  
  Rest of the partitions  
  normal: Supported for both zynq and zynqmp. Splits as follows:  
  Bootheader + Image Headers + Partition Headers + Bootloader  
  Partition1  
  Partition2 and so on  
  Along with the split mode, output format can also be specified as bin or mcs.  
  Note: The option split mode normal is same as the command line option split. This command line option is deprecated. | zynq, zynqmp |
| spk_select <SPK_ID> | Specify an SPK ID in user eFUSE. | zynqmp |
| spksignature <signature_file> | Imports the SPK signature into an Authentication Certificate. See Using Authentication. This can be used in case the user does not want to share the secret key PSK. The user can create a signature and provide it to Bootgen. | zynq, zynqmp |
| sskfile <key filename> | Secondary Secret Key (SSK) key authenticates partitions in the Boot Image. The primary keys authenticate the secondary keys; the secondary keys authenticate the partitions. | All |
| startup=<address> | Sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute. | zynq, zynqmp |
| trustzone= <option> | The trustzone options are:  
  secure  
  nonsecure | zynqmp |
| udf_bh <data_file> | Imports a file of data to be copied to the user defined field (UDF) of the Boot Header. The UDF is provided through a text file in the form of a hex string. Total number of bytes in UDF are: zynq = 76 bytes; zynqmp= 40 bytes. | zynq, zynqmp |
| udf_data <data_file> | Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate. | zynq, zynqmp |
| xip_mode | Indicates eXecute In Place (XIP) for FSBL to be executed directly from QSPI flash. | zynq, zynqmp |
Using Bootgen Interfaces

Bootgen has both a GUI and a command line option. The GUI option is available in the Vitis IDE as a wizard. The functionality in this GUI is limited to the most standard functions when creating a boot image. The bootgen command line, however, is a full-featured set of commands that lets you create a complex boot image for your system.

Bootgen GUI Options

The Create Boot Image wizard offers a limited number of Bootgen options to generate a boot image.

To create a boot image using the GUI, do the following:

1. Select the application project in the Project Navigator or C/C++ Projects view and right-click Create Boot Image. Alternatively, click Xilinx → Create Boot Image.

   The Create Boot Image dialog box opens, with default values pre-selected from the context of the selected C project.

   Note the following:
   - When you run Create Boot Image the first time for an application, the dialog box is pre-populated with paths to the FSBL ELF file, and the bitstream for the selected hardware (if it exists in hardware project), and then the selected application ELF file.
   - If a boot image was run previously for the application, and a BIF file exists, the dialog box is pre-populated with the values from the /bif folder.
   - You can now create a boot image for Zynq®-7000 SoC or Zynq® UltraScale+™ MPSoC architectures.

   ✤ IMPORTANT! The data you enter for the boot image should be a maximum of 76 bytes with an offset of 0x4c (for Zynq-7000 SoC) and 40 bytes and an offset of 0x70 (for Zynq UltraScale+ MPSoC). This is a hard limitation based on the Zynq architecture.

2. Populate the Create boot image dialog box with the following information:
   a. From the Architecture drop-down, select the required architecture.
   b. Select either Create a BIF file or Import an existing BIF file.
c. From the Basic tab, specify the **Output BIF file path**.
d. If applicable, specify the **UDF data**: See `udf_data` for more information about this option.
e. Specify the **Output path**:

3. In the Boot image partitions, click the **Add** button to add additional partition images.

4. Create offset, alignment, and allocation values for partitions in the boot image, if applicable.
   
   The output file path is set to the `/bif` folder under the selected application project by default.

5. From the Security tab, you can specify the attributes to create a secure image. This security can be applied to individual partitions as required.
   
   a. To enable **Authentication** for a partition, check the **Use Authentication** option, then specify the PPK, SPK, PSK, and SSK values. See the **Authentication** topic for more information.
   
   b. To enable **Encryption** for a partition, select the Encryption tab, and check the **Use Encryption** option. See **Using Encryption** for more information.

6. Create or import a BIF file boot image one partition at a time, starting from the bootloader. The partitions list displays the summary of the partitions in the BIF file. It shows the file path, encryption settings, and authentication settings. Use this area to add, delete, modify, and reorder the partitions. You can also set values for enabling encryption, authentication, and checksum, and specifying some other partition related values like **Load**, **Alignment**, and **Offset**.

---

### Using Bootgen on the Command Line

When you specify Bootgen options on the command line you have many more options than those provided in the GUI. In the standard install of the Vitis software platform, the XSCT (Xilinx Software Command-Line Tool) is available for use as an interactive command line environment, or to use for creating scripting. In the XSCT, you can run Bootgen commands. XSCT accesses the Bootgen executable, which is a separate tool. This bootgen executable can be installed stand-alone as described in **Installing Bootgen**. This is the same tool as is called from the XSCT, so any scripts developed here or in the XSCT will work in the other tool.

The **Xilinx Software Command-Line Tools (XSCT) Reference Guide (UG1208)** describes the tool. See the XSCT Use Cases chapter for an example of using Bootgen commands in XSCT.
## Commands and Descriptions

The following table lists the Bootgen command options. Each option is linked to a longer description in the left column with a short description in the right column. The architecture name indicates what Xilinx® device uses that command:

- **zynq**: Zynq®-7000 SoC device
- **zynqmp**: Zynq® UltraScale+™ MPSoC device
- **fpga**: Any 7 series and above devices

### Table 20: Bootgen Command and Descriptions

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description and Options</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arch &lt;type&gt;</code></td>
<td>Xilinx® device architecture: Options:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>- zynq (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- zynqmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- fpga</td>
<td></td>
</tr>
<tr>
<td><code>bif_help</code></td>
<td>Prints out the BIF help summary.</td>
<td>All</td>
</tr>
<tr>
<td><code>dual_qspi_mode &lt;configuration&gt;</code></td>
<td>Generates two output files for dual QSPI configurations:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- parallel</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>- stacked &lt;size&gt;</td>
<td>zynqmp</td>
</tr>
<tr>
<td><code>efuseppkbits &lt;PPK_filename&gt;</code></td>
<td>Generates a PPK hash for eFUSE.</td>
<td></td>
</tr>
<tr>
<td><code>encrypt &lt;options&gt;</code></td>
<td>AES Key storage in device. Options are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- bbram (default)</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>- efuse</td>
<td>fpga</td>
</tr>
<tr>
<td><code>encryption_dump</code></td>
<td>Generates encryption log file, aes_log.txt.</td>
<td></td>
</tr>
<tr>
<td><code>fill &lt;hex_byte&gt;</code></td>
<td>Specifies the fill byte to use for padding.</td>
<td></td>
</tr>
<tr>
<td><code>generate_hashes</code></td>
<td>Output SHA2/SHA3 hash files with padding in PKCS#1v1.5 format.</td>
<td></td>
</tr>
<tr>
<td><code>generate_keys &lt;key_type&gt;</code></td>
<td>Generate the authentication keys. Options are:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- pem</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>- rsa</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>- obfuscatedkey</td>
<td></td>
</tr>
</tbody>
</table>
### Table 20: Bootgen Command and Descriptions (cont'd)

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description and Options</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>h, help</td>
<td>Prints out help summary.</td>
<td>All</td>
</tr>
<tr>
<td>image &lt;filename(.bif)&gt;</td>
<td>Provides a boot image format (.bif) file name.</td>
<td>All</td>
</tr>
</tbody>
</table>
| log<level_type> | Generates a log file at the current working directory with following message types:  
|               | error  
|               | warning (default)  
|               | info  
|               | debug  
|               | trace  | All         |
| nonbooting     | Create an intermediate boot image.                   | zynq        |
|                |                                                     | zynqmp      |
| o <filename>   | Specifies the output file. The format of the file is determined by the filename extension. Valid extensions are:  
|                | .bin (default)  
|                | .mcs  | All         |
| p <partname>   | Specify the part name used in generating the encryption key. | All         |
| padimageheader | Pads the image headers to force alignment of following partitions. Options are:  
| <option>       | 0  
|                | 1 (default)  | zynq        |
|                |                                                     | zynqmp      |
| process_bitstream | Specifies that the bitstream is processed and outputs as .bin or .mcs.  
| <option>       | For example, if encryption is selected for bitstream in BIF file, the output is an encrypted bitstream. | zynq        |
|                |                                                     | zynqmp      |
| read <options> | Used to read boot headers, image headers, and partition headers based on the options.  
|                | bh: To read boot header from bootimage in human readable form  
|                | iht: To read image header table from bootimage  
|                | ih: To read image headers from bootimage  
|                | pht: To read partition headers from bootimage  
<p>|                | ac: To read authentication certificates from bootimage | zynq        |
|                |                                                     | zynqmp      |</p>
<table>
<thead>
<tr>
<th>Commands</th>
<th>Description and Options</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>split &lt;options&gt;</code></td>
<td>Splits the boot image into partitions and outputs the files as <code>.bin</code> or <code>.mcs</code>.</td>
<td>zynq</td>
</tr>
<tr>
<td></td>
<td>• Bootheader + Image Headers + Partition Headers + Fsbl.elf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Partition1.bit</td>
<td>zynqmp</td>
</tr>
<tr>
<td></td>
<td>• Partition2.elf</td>
<td></td>
</tr>
<tr>
<td><code>spksignature &lt;filename&gt;</code></td>
<td>Generates an SPK signature file.</td>
<td>zynq</td>
</tr>
<tr>
<td><code>verify</code></td>
<td>This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image will be verified against the available partitions.</td>
<td>zynq</td>
</tr>
<tr>
<td><code>verify_kdf</code></td>
<td>This option is used to validate the Counter Mode KDF used in bootgen for generation AES keys.</td>
<td>zynqmp</td>
</tr>
<tr>
<td><code>w &lt;option&gt;</code></td>
<td>Specifies whether to overwrite the output files:</td>
<td>All</td>
</tr>
<tr>
<td></td>
<td>on (default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>off</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The <code>-w</code> without an option is interpreted as <code>-w on</code>.</td>
<td></td>
</tr>
<tr>
<td><code>zynqmpes1</code></td>
<td>Generates a boot image for ES1 (1.0). The default padding scheme is ES2 (2.0).</td>
<td>zynqmp</td>
</tr>
</tbody>
</table>
Boot Time Security

Xilinx® supports secure booting on all devices using latest authentication methods to prevent unauthorized or modified code from being run on Xilinx devices. Xilinx supports various encryption techniques to make sure only authorized programs access the images. For hardware security features by device, see the following sections.

Secure and Non-Secure Modes in Zynq-7000 SoC Devices

For security reasons, CPU 0 is always the first device out of reset among all master modules within the PS. CPU 1 is held in an WFE state. While the BootROM is running, the JTAG is always disabled, regardless of the reset type, to ensure security. After the BootROM runs, JTAG is enabled if the boot mode is non-secure.

The BootROM code is also responsible for loading the FSBL/User code. When the BootROM releases control to stage 1, the user software assumes full control of the entire system. The only way to execute the BootROM again is by generating one of the system resets. The FSBL/User code size, encrypted and unencrypted, is limited to 192 KB. This limit does not apply with the non-secure execute-in-place option.

The PS boot source is selected using the BOOT_MODE strapping pins (indicated by a weak pull-up or pull-down resistor), which are sampled once during power-on reset (POR). The sampled values are stored in the slcr.BOOT_MODE register.

The BootROM supports encrypted/authenticated, and unencrypted images referred to as secure boot and non-secure boot, respectively. The BootROM supports execution of the stage 1 image directly from NOR or Quad-SPI when using the execute-in-place (xip_mode) option, but only for non-secure boot images. Execute-in-place is possible only for NOR and Quad-SPI boot modes.

- In secure boot, the CPU, running the BootROM code decrypts and authenticates the user PS image on the boot device, stores it in the OCM, and then branches to it.
- In non-secure boot, the CPU, running the BootROM code disables all secure boot features including the AES unit within the PL before branching to the user image in the OCM memory or the flash device (if execute-in-place (XIP) is used).

Any subsequent boot stages for either the PS or the PL are the responsibility of you, the developer, and are under your control. The BootROM code is not accessible to you. Following a stage 1 secure boot, you can proceed with either secure or non-secure subsequent boot stages. Following a non-secure first stage boot, only non-secure subsequent boot stages are possible.
**Zynq UltraScale+ MPSoC Device Security**

In a Zynq® UltraScale+™ MPSoC device, the secure boot is accomplished by using the hardware root of trust boot mechanism, which also provides a way to encrypt all of the boot or configuration files. This architecture provides the required confidentiality, integrity, and authentication to host the most secure of applications.

See this link in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for more information.

---

**Using Encryption**

Secure booting, which validates the images on devices before they are allowed to execute, has become a mandatory feature for most electronic devices being deployed in the field. For encryption, Xilinx supports an advanced encryption standard (AES) algorithm AES encryption.

AES provides symmetric key cryptography (one key definition for both encryption and decryption). The same steps are performed to complete both encryption and decryption in reverse order.

AES is an iterated symmetric block cipher, which means that it does the following:

- Works by repeating the same defined steps multiple times
- Uses a secret key encryption algorithm
- Operates on a fixed number of bytes

**Encryption Process**

Bootgen can encrypt the boot image partitions based on the user-provided encryption commands and attributes in the BIF file. AES is a symmetric key encryption technique; it uses the same key for encryption and decryption. The key used to encrypt a boot image should be available on the device for the decryption process while the device is booting with that boot image. Generally, the key is stored either in eFUSE or BBRAM, and the source of the key can be selected during boot image creation through BIF attributes, as shown in the following figure.
Decryption Process

For SoC devices, the BootROM and the FSBL decrypt partitions during the booting cycle. The BootROM reads FSBL from flash, decrypts, loads, and hands off the control. After FSBL start executing, it reads the remaining partitions, decrypts, and loads them. The AES key needed to decrypt the partitions can be retrieved from either eFUSE or BBRAM. The key source field of the Boot Header table in the boot image is read to know the source of the encryption key. Each encrypted partition is decrypted using a AES hardware engine.
Encrypting Zynq-7000 Device Partitions

Zynq-7000 SoC devices use the embedded, Progammable Logic (PL), hash-based message authentication code (HMAC) and an advanced encryption standard (AES) module with a cipher block chaining (CBC) mode.

Example BIF File

To create a boot image with encrypted partitions, the AES key file is specified in the BIF using the `aeskeyfile` attribute. Specify an `encryption=aes` attribute for each image file listed in the BIF file to be encrypted. The example BIF file (`secure.bif`) is shown below:

```
image:
[
    [aeskeyfile] secretkey.nky
    [keysrc_encryption] efuse
    [bootloader, encryption=aes] fsbl.elf
    [encryption=aes] uboot.elf
]
```

From the command line, use the following command to generate a boot image with encrypted `fsbl.elf` and `uboot.elf`.

```
bootgen -arch zynq -image secure.bif -w -o BOOT.bin
```

Key Generation

Bootgen can generate AES-CBC keys. Bootgen uses the AES key file specified in the BIF for encrypting the partitions. If the key file is empty or non-existent, Bootgen generates the keys in the file specified in the BIF file. If the key file is not specified in the BIF, and encryption is requested for any of the partitions, then Bootgen generates a key file with the name of the BIF file with extension `.nky` in the same directory as of BIF. The following is a sample key file.

![Figure 7: Sample Key File](image)

Encrypting Zynq MPSoC Device Partitions

The Zynq UltraScale+ MPSoC device uses the AES-GCM core, which has a 32-bit, word-based data interface with support for a 256-bit key. The AES-GCM mode supports encryption and decryption, multiple key sources, and built-in message integrity check.
Operational Key

A good key management practice includes minimizing the use of secret or private keys. This can be accomplished using the operational key option enabled in Bootgen.

Bootgen creates an encrypted, secure header that contains the operational key (\texttt{opt\_key}), which is user-specified, and the initialization vector (IV) needed for the first block of the configuration file when this feature is enabled. The result is that the AES key stored on the device, in either the BBRAM or eFUSEs, is used for only 384 bits, which significantly limits its exposure to side channel attacks. The attribute \texttt{opt\_key} is used to specify operational key usage. See \texttt{fsbl\_config} for more information about the \texttt{opt\_key} value that is an argument to the \texttt{fsbl\_config} attribute. The following is an example of using the \texttt{opt\_key} attribute.

```plaintext
image:
{
  [fsbl\_config] opt\_key
  [keysrc\_encryption] bbram\_red\_key

  [bootloader,
   destination\_cpu = a53-0,
   encryption = aes,
   aeskeyfile = aes\_p1\_nky|fsbl.elf
  ]

  [destination\_cpu = a53-3,
   encryption = aes,
   aeskeyfile = aes\_p2\_nky|hello.elf
  ]
}
```

The operation key is given in the AES key (.nky) file with name \texttt{Key Opt} as shown in the following example.

![Figure 8: Operational Key](image)

Bootgen generates the encryption key file. The operational key \texttt{opt\_key} is then generated in the .nky file, if \texttt{opt\_key} has been enabled in the BIF file, as shown in the previous example.

For another example of using the operational key, refer to Using Op Key to Protect the Device Key in a Development Environment.

For more details about this feature, see the Key Management section of the "Security" chapter in the Zynq UltraScale+ Device Technical Reference Manual (UG1085).
**Rolling Keys**

The AES-GCM also supports the rolling keys feature, where the entire encrypted image is represented in terms of smaller AES encrypted blocks/modules. Each module is encrypted using its own unique key. The initial key is stored at the key source on the device, while keys for each successive module are encrypted (wrapped) in the previous module. The boot images with rolling keys can be generated using Bootgen. The BIF attribute `blocks` is used to specify the pattern to create multiple smaller blocks for encryption.

```plaintext
image:
{
    [keysrc_encryption] bbram_red_key
    
    
    
    
    bootloader,
        destination_cpu = a53-0,
        encryption = aes,
        aeskeyfile = aes_p1.nky,
        blocks = 1024(2);2048;4096(2);8192(2);4096;2048;1024
    }  fsbl.elf

    
    
    
    
    destination_cpu = a53-3,
        encryption = aes,
        aeskeyfile = aes_p2.nky,
        blocks = 4096(1);1024
    }  hello.elf
}
```

**Note:**

- Number of keys in the key file should always be equal to the number of blocks to be encrypted.
  - If the number of keys are less than the number of blocks to be encrypted, Bootgen returns an error.
  - If the number of keys are more than the number of blocks to be encrypted, Bootgen ignores the extra keys.
- If you want to specify multiple Key/IV Pairs, you should specify `no. of blocks + 1` pairs
  - The extra Key/IV pair is to encrypt the secure header.

**Gray/Obfuscated Keys**

The user key is encrypted with the family key, which is embedded in the metal layers of the device. This family key is the same for all devices in the Zynq® UltraScale+™ MPSoC. The result is referred to as the `obfuscated key`. 
The obfuscated key can reside in either the Authenticated Boot Header or in eFUSEs.

```
image:
[
  [keysrc_encryption] efuse_gry_key
  [bh_key_iv] bhiv.txt
  [bootloader
    destination_cpu = a53-0,
    encryption = aes,
    aeskeyfile = aes_p1.nky
  ]
  fsbl.elf
]
[
  destination_cpu = r5-0,
  encryption = aes,
  aeskeyfile = aes_p2.nky
]
hello.elf
]
```

Bootgen does the following while creating an image:

1. Places the IV from `bhiv.txt` in the field *BH IV* in Boot Header.
2. Places the IV 0 from `aes.nky` in the field "Secure Header IV" in Boot Header.
3. Encrypts the partition, with Key0 and IV0 from `aes.nky`.

Another example of using the gray/family key is found in Appendix A: Use Cases and Examples.

For more details about this feature, refer to the Zynq UltraScale+ Device Technical Reference Manual (UG1085).

**Key Generation**

Bootgen has the capability of generating AES-GCM keys. It uses the NIST-approved Counter Mode KDF, with CMAC as the pseudo random function. Bootgen takes seed as input in case the user wants to derive multiple keys from seed due to key rolling. If a seed is specified, the keys are derived using the seed. If seeds are not specified, keys are derived based on Key0. If an empty key file is specified, Bootgen generates a seed with time based randomization (not KDF), which in turn is the input for KDF to generate other the Key/IV pairs.

**Note:**

- If one encryption file is specified and others are generated, Bootgen can make sure to use the same Key0/IV0 pair for the generated keys as in the encryption file for first partition.
- If an encryption file is generated for the first partition and other encryption file with Key0/IV0 is specified for a later partition, then Bootgen exits and returns the error that an incorrect Key0/IV0 pair was used.

**Key Generation**

A sample key file is shown below.
Obfuscated Key Generation

Bootgen can generate the Obfuscated key by encrypting the red key with the family key and a user-provided IV. The family key is delivered by the Xilinx® Security Group. For more information, see `familykey`. To generate an obfuscated key, Bootgen takes the following inputs from the BIF file.

```plaintext
obf_key:
[
    [aeskeyfile] aes.nky
    [familykey] familyKey.cfg
    [bh_key_iv] bhiv.txt
]
```

The command to generate the Obfuscated key is:

```bash
bootgen -arch zynqmp -image all.bif -generate_keys obfuscatedkey
```

Black/PUF Keys

The black key storage solution uses a cryptographically strong key encryption key (KEK), which is generated from a PUF, to encrypt the user key. The resulting black key can then be stored either in the eFUSE or as a part of the authenticated boot header.

```plaintext
image:
[
    [puf_file] pufdata.txt
    [bh_key_iv] black_iv.txt
    [bh_keyfile] black_key.txt
    [fsbl_config] puf4kmode, shutter=0x0100005E, pufhd_bh
    [keysrc_encryption] bh_blk_key
    [
        bootloader,
        destination_cpu = a53-0,
        encryption     = aes,
        aeskeyfile      = aes_p1.nky
    ] fsbl.elf
]
```
destination_cpu = r5-0,
encryption = aes,
aeskeyfile = aes_p2.nky
] hello.elf
]

For another example of using the black key, see Appendix A: Use Cases and Examples.

**Multiple Encryption Key Files**

Earlier versions of Bootgen supported creating the boot image by encrypting multiple partitions with a single encryption key. The same key is used over and over again for every partition. This is a security weakness and not recommended. Each key should be used only once in the flow.

Bootgen supports separate encryption keys for each partition. In case of multiple key files, ensure that each encryption key file uses the same Key0 (device key), IVO, and Operational Key. Bootgen does not allow creating boot images if these are different in each encryption key file. You must specify multiple encryption key files, one for each of partition in the image. The partitions are encrypted using the key that is specified for the partition.

**Note:** You can have unique key files for each of the partition created due to multiple loadable sections by having key file names appended with ".1", ".2", ..., ".n" so on in the same directory of the key file meant for that partition.

The following snippet shows a sample encryption key file:

```plaintext
all:
[
  [keysrc_encryption] bbram_red_key
  // FSBL (Partition-0)
  [bootsloader, destination_cpu = a53-0, encryption = aes, aeskeyfile = key_p0.nky
   ] fsb1a53.elf
  // application (Partition-1)
  [destination_cpu = a53-0, encryption = aes, aeskeyfile = key_p1.nky
   ] hello.elf
]
```

- **The partition** `fsb1a53.elf` is encrypted using the keys from `key_p0.nky` file.
- **Assuming** `hello.elf` has three partitions because it has three loadable sections, then partition `hello.elf.0` is encrypted using keys from the `test2.nky` file.
- **Partition** `hello.elf.1` is then encrypted using keys from `test2.1.nky`.
- **Partition** `hello.elf.2` is encrypted using keys from `test2.2.nky`. 
Using Authentication

AES encryption is a self-authenticating algorithm with a symmetric key, meaning that the key to encrypt is the same as the one to decrypt. This key must be protected as it is secret (hence storage to internal key space). There is an alternative form of authentication in the form of RSA (Rivest-Shamir-Adleman). RSA is an asymmetric algorithm, meaning that the key to verify is not the same key used to sign. A pair of keys are needed for authentication.

- Signing is done using Secret Key/ Private Key
- Verification is done using a Public Key

This public key does not need to be protected, and does not need special secure storage. This form of authentication can be used with encryption to provide both authenticity and confidentiality. RSA can be used with either encrypted or unencrypted partitions.

RSA not only has the advantage of using a public key, it also has the advantage of authenticating prior to decryption. The hash of the RSA Public key must be stored in the eFUSE. Xilinx® SoC devices support authenticating the partition data before it is sent to the AES decryption engine. This method can be used to help prevent attacks on the decryption engine itself by ensuring that the partition data is authentic before performing any decryption.

In Xilinx SoCs, two pairs of public and secret keys are used - primary and secondary. The function of the primary public/secret key pair is to authenticate the secondary public/secret key pair. The function of the secondary key is to sign/verify partitions.

The first letter of the acronyms used to describe the keys is either P for primary or S for secondary. The second letter of the acronym used to describe the keys is either P for public or S for secret. There are four possible keys:

- PPK = Primary Public Key
- PSK = Primary Secret Key
- SPK = Secondary Public Key
- SSK = Secondary Secret Key

Bootgen can create an authentication certificate in two ways:

- Supply the PSK and SSK. The SPK signature is calculated on-the-fly using these two inputs.
- Supply the PPK and SSK and the SPK signature as inputs. This is used in cases where the PSK is not known.

The primary key is hashed and stored in the eFUSE. This hash is compared against the hash of the primary key stored in the boot image by the FSBL. This hash can be written to the PS eFUSE memory using standalone driver provided along with Vitis.
The following is an example BIF file:

```plaintext
image:
[
  [pskfile]primarykey.pem
  [sskfile]secondarykey.pem
  [bootloader,authentication=rsa] fsbl.elf
  [authentication=rsa]uboot.elf
]
```

For device-specific Authentication information, see the following:

- Zynq-7000 Authentication Certificates
- Zynq UltraScale+ MPSoC Authentication Certificates

**Signing**

The following figure shows RSA signing of partitions. From a secure facility, Bootgen signs partitions using the Secret key. The signing process is described in the following steps:

1. PPK and SPK are stored in the Authentication Certificate (AC).
2. SPK is signed using PSK to get SPK signature; also stored as part of the AC.
3. Partition is signed using SSK to get Partition signature, populated in the AC.
4. The AC is appended to each partition that is opted for authentication.
5. PPK is hashed and stored in eFUSE.
The following table shows the options for Authentication.

**Table 21: Supported File Formats for Authentication Keys**

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Description</th>
<th>Supported File Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPK</td>
<td>Primary Public Key</td>
<td>This key is used to authenticate a partition. It should always be specified when authenticating a partition.</td>
<td>*.txt, *.pem, *.pub, *.pk1</td>
</tr>
<tr>
<td>PSK</td>
<td>Primary Secret Key</td>
<td>This key is used to authenticate a partition. It should always be specified when authenticating a partition.</td>
<td>*.txt, *.pem, *.pk1</td>
</tr>
<tr>
<td>SPK</td>
<td>Secondary Public Key</td>
<td>This key, when specified, is used to authenticate a partition.</td>
<td>*.txt, *.pem, *.pub, *.pk1</td>
</tr>
</tbody>
</table>
Table 21: Supported File Formats for Authentication Keys (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Description</th>
<th>Supported File Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSK</td>
<td>Secondary Secret Key</td>
<td>This key, when specified, is used to authenticate a partition.</td>
<td>*.txt, *.pem, *.pk1</td>
</tr>
</tbody>
</table>

Verifying

In the device, the BootROM verifies the FSBL, and either the FSBL or U-Boot verifies the subsequent partitions using the Public key.

1. **Verify PPK** - This step establishes the authenticity of primary key, which is used to authenticate secondary key.
   a. PPK is read from AC in boot image
   b. Generate PPK hash
   c. Hashed PPK is compared with the PPK hash retrieved from eFUSE
   d. If same, then primary key is trusted, else secure boot fail

2. **Verify secondary keys** - This step establishes the authenticity of secondary key, which is used to authenticate the partitions.
   a. SPK is read from AC in boot image
   b. Generate SPK hashed
   c. Get the SPK hash, by verifying the SPK signature stored in AC, using PSK
   d. Compare hashes from step (b) and step (c)
   e. If same, then secondary key is trusted, else secure boot fail.

3. **Verify partitions** - This step establishes the authenticity of partition which is being booted.
   a. Partition is read from the boot image.
   b. Generate hash of the partition.
   c. Get the partition hash, by verifying the Partition signature stored in AC, using SSK.
   d. Compare the hashes from step (b) and step (c)
   e. If same, then partition is trusted, else secure boot fail.
Bootgen can create an authentication certificate in two ways:

- Supply the PSK and SSK. The SPK signature is calculated on-the-fly using these two inputs.
- Supply the PPK and SSK and the SPK signature as inputs. This is used in cases where the PSK is not known.

### Zynq UltraScale+ MPSoC Authentication Support

The Zynq® UltraScale+™ MPSoC device uses RSA-4096 authentication, which means the primary and secondary key sizes are 4096-bit.

### NIST SHA-3 Support

**Note:** For SHA-3 Authentication, always use Keccak SHA-3 to calculate hash on boot header, PPK hash and boot image. NIST-SHA3 is used for all other partitions which are not loaded by ROM.

The generated signature uses the Keccak-SHA3 or NIST-SHA3 based on following table:
### Table 22: Authentication Signatures

<table>
<thead>
<tr>
<th>Which Authentication Certificate (AC)?</th>
<th>Signature</th>
<th>SHA Algorithm and SPK eFUSE</th>
<th>Secret Key used for Signature Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header AC (loader by FSBL/FW)</td>
<td>SPK Signature</td>
<td>If SPKID eFUSEs, then Keecak; If User eFUSE, then NIST</td>
<td>PSK</td>
</tr>
<tr>
<td></td>
<td>BH Signature</td>
<td>Always Keecak</td>
<td>SSK&lt;sub&gt;header&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>Header Signature</td>
<td>Always Nist</td>
<td>SSK&lt;sub&gt;header&lt;/sub&gt;</td>
</tr>
<tr>
<td>BootLoader AC (loaded by ROM)</td>
<td>SPK Signature</td>
<td>Always Keecak; Always SPKID eFUSE for SPK</td>
<td>PSK</td>
</tr>
<tr>
<td></td>
<td>BH Signature</td>
<td>Always Keecak</td>
<td>SSK&lt;sub&gt;bootloader&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>Header Signature</td>
<td>Always Keecak</td>
<td>SSK&lt;sub&gt;bootloader&lt;/sub&gt;</td>
</tr>
<tr>
<td>Partition AC (loaded by FSBL FW)</td>
<td>SPK Signature</td>
<td>If SPKID eFUSEs then Keecak; If User eFUSE then NIST</td>
<td>PSK</td>
</tr>
<tr>
<td></td>
<td>BH Signature</td>
<td>Always Keecak</td>
<td>SSK&lt;sub&gt;partition&lt;/sub&gt;</td>
</tr>
<tr>
<td></td>
<td>Header Signature</td>
<td>Always NIST</td>
<td>SSK&lt;sub&gt;partition&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

### Examples

Example 1: BIF file for authenticating the partition with single set of key files:

```plaintext
image:
{
    [fsbl_config] bh_auth_enable
    [auth_params] ppk_select=0; spk_id=0x00000000
    [pskfile] primary_4096.pem
    [sskfile] secondary_4096.pem
    [pmufw_image] pmufw.elf
    [bootloader, authentication=rsa, destination_cpu=a53-0] fsbl.elf
    [authentication=rsa, destination_cpu=r5-0] hello.elf
}
```
Example 2: BIF file for authenticating the partitions with separate secondary key for each partition:

image:
{
    [auth_params] ppk_select=1
    [pskfile] primary_4096.pem
    [sskfile] secondary_4096.pem

    // FSBL (Partition-0)
    [bootloader,
    destination_cpu = a53-0,
    authentication = rsa,
    spk_id = 0x01,
    sskfile = secondary_pl.pem
    ] fsbla53.elf

    // ATF (Partition-1)
    [destination_cpu = a53-0,
    authentication = rsa,
    exception_level = el-3,
    trustzone = secure,
    spk_id = 0x01,
    sskfile = secondary_p2.pem
    ] bl31.elf

    // UBOOT (Partition-2)
    [destination_cpu = a53-0,
    authentication = rsa,
    exception_level = el-2,
    spk_id = 0x01,
    sskfile = secondary_p3.pem
    ] u-boot.elf
}

**Bitstream Authentication Using External Memory**

The authentication of a bitstream is different from other partitions. The FSBL can be wholly contained within the OCM, and therefore authenticated and decrypted inside of the device. For the bitstream, the size of the file is so large that it cannot be wholly contained inside the device and external memory must be used. The use of external memory creates a challenge to maintain security because an adversary may have access to this external memory. When bitstream is requested for authentication, Bootgen divides the whole bitstream into 8MB blocks and has an authentication certificate for each block. If a bitstream is not in multiples of 8MB, the last block contains the remaining bitstream data. When authentication and encryption are both enabled, encryption is first done on the bitstream, then Bootgen divides the encrypted data into blocks and places an authentication certificate for each block.
User eFUSE Support with Enhanced RSA Key Revocation

Enhanced RSA Key Revocation Support

The RSA key provides the ability to revoke the secondary keys of one partition without revoking the secondary keys for all partitions.

*Note:* The primary key should be the same across all partitions.

This is achieved by using `USER_FUSE0` to `USER_FUSE7` eFUSEs with the BIF parameter `spk_select`.

*Note:* You can revoke up to 256 keys, if all are not required for their usage.

The following BIF file sample shows enhanced user fuse revocation. Image header and FSBL uses different SSKs for authentication (`ssk1.pem` and `ssk2.pem` respectively) with the following BIF input.

```python
the_ROM_image:
{
    [auth_params]ppk_select = 0
    [pskfile]psk.pem
    [sskfile]ssk1.pem
    [bootloader,
     authentication = rsa,
     spk_select = spk-efuse,
     spk_id = 0x8,
     sskfile = ssk2.pem
    ] zynqmp_fsbl.elf
    [destination_cpu = a53-0,
     authentication = rsa,
     spk_select = user-efuse,
     spk_id = 0x200,
     sskfile = ssk3.pem
    ]
}
```
application.elf
{
    destination_cpu = a53-0,
    authentication = rsa,
    spk_select = spk-efuse,
    spk_id = 0x8,
    sskfile = ssk4.pem
} application2.elf

- `spk_select = spk-efuse` indicates that spk_id eFUSE will be used for that partition.
- `spk_select = user-efuse` indicates that user eFUSE will be used for that partition.

Partitions loaded by CSU ROM will always use `spk_efuse`.

_Note:_ The `spk_id` eFUSE specifies which key is valid. Hence, the ROM checks the entire field of `spk_id` eFUSE against the SPK ID to make sure its a bit for bit match.

The user eFUSE specifies which key ID is NOT valid (has been revoked). Therefore, the firmware (non-ROM) checks to see if a given user eFUSE that represents the SPK ID has been programmed.

### Key Generation

Bootgen has the capability of generating RSA keys. Alternatively, you can create keys using external tools such as OpenSSL. Bootgen creates the keys in the paths specified in the BIF file.

The figure shows the sample RSA private key file.
BIF Example

A sample BIF file, `generate_pem.bif`:

```plaintext
generate_pem:
{
  [pskfile] psk0.pem
  [sskfile] ssk0.pem
}
```
Command

The command to generate keys is, as follows:

```bash
bootgen -generate_keys pem -arch zynqmp -image generate_pem.bif
```

PPK Hash for eFUSE

Bootgen generates the PPK hash for storing in eFUSE for PPK to be trusted. This step is required only for RSA Authentication with eFUSE mode, and can be skipped for RSA Boot Header Authentication for the Zynq® UltraScale+™ MPSoC device. The value from `efuseppksha.txt` can be programmed to eFUSE for RSA authentication with the eFUSE mode.

For more information about BBRAM and eFUSE programming, see *Programming BBRAM and eFUSES* (*XAPP1319*).

BIF File Example

The following is a sample BIF file, `generate_hash_ppk.bif`.

```plaintext
generate_hash_ppk:
{
  [pskfile] psk0.pem
  [sskfile] ssk0.pem
  [bootloader, destination_cpu=a53-0, authentication=rsa] fsbl_a53.elf
}
```

Command

The command to generate PPK hash for eFUSE programming is:

```bash
bootgen –image generate_hash_ppk.bif –arch zynqmp –w –o /test.bin –efuseppkbits efuseppksha.txt
```

Using HSM Mode

In current cryptography, all the algorithms are public, so it becomes critical to protect the private/secret key. The hardware security module (HSM) is a dedicated crypto-processing device that is specifically designed for the protection of the crypto key lifecycle, and increases key handling security, because only public keys are passed to the Bootgen and not the private/secure keys. A *Standard* mode is also available; this mode does not require passing keys.
In some organizations, an Infosec staff is responsible for the production release of a secure embedded product. The Infosec staff might use a HSM for digital signatures and a separate secure server for encryption. The HSM and secure server typically reside in a secure area. The HSM is a secure key/signature generation device which generates private keys, signs the partitions using the private key, and provides the public part of the RSA key to Bootgen. The private keys reside in the HSM only.

Bootgen in HSM mode uses only RSA public keys and the signatures that were created by the HSM to generate the boot image. The HSM accepts hash values of partitions generated by Bootgen and returns a signature block, based on the hash and the secret RSA key.

In contrast to the HSM mode, Bootgen in its Standard mode uses AES encryption keys and the RSA Secret keys provided through the BIF file, to encrypt and authenticate the partitions in the image, respectively. The output is a single boot image, which is encrypted and authenticated. For authentication, the user has to provide both sets of public and private/secret keys. The private/secret keys are used by the Bootgen to sign the partitions and create signatures. These signatures along with the public keys are embedded into the final boot image.

For more information about the HSM mode for FPGAs, see the HSM Mode for FPGAs.

Using Advanced Key Management Options

The public keys associated with the private keys are ppk.pub and spk.pub. The HSM accepts hash values of partitions generated by Bootgen and returns a signature block, based on the hash and the secret key.

Creating a Boot Image Using HSM Mode: PSK is not Shared

The following figure shows a Stage 0 to Stage 2 Boot stack that uses the HSM mode. It reduces the number of steps by distributing the SSK.

This figure uses the Zynq® UltraScale+™ MPSoC device to illustrate the stages.
Boot Process

Creating a boot image using HSM mode is similar to creating a boot image using a Standard flow with following BIF file.

```plaintext
all:
{
    [auth_params] ppk_select=1;spk_id=0x12345678
    [keysrc_encryption]bbram_red_key
    [pskfile]primary.pem
    [sskfile]secondary.pem
    [bootloader,
      encryption=aes,
      aeskeyfile=aes.nky,
      authentication=rsa
    ]fsbl.elf
    [destination_cpu=a53-0,authentication=rsa]hello_a53_0_64.elf
}
```

Stage 0: Create a boot image using HSM Mode

A trusted individual creates the SPK signature using the Primary Secret Key. The SPK Signature is on the Authentication Certificate Header, SPK, and SPKID. Generate a hash for SPK. The following is the snippet from the BIF file.

```plaintext
stage 0:
{
    [auth_params] ppk_select=1;spk_id=0x12345678
    [spkfile]keys/secondary.pub
}
```
The following is the Bootgen command:

```
bootgen -arch zynqmp -image stage0.bif -generate_hashes
```

The output of this command is: `secondary.pub.sha384`.

**Stage 1: Distribute the SPK Signature**

The trusted individual distributes the SPK Signature to the development teams.

```
openssl rsautl -raw -sign -inkey keys/primary0.pem -in secondary.pub.sha384 > secondary.pub.sha384.sig
```

The output of this command is: `secondary.pub.sha384.sig`.

**Stage 2: Encrypt using AES in FSBL**

The development teams use Bootgen to create as many boot images as needed. The development teams use:

- The SPK Signature from the Trusted Individual.
- The Secondary Secret Key (SSK), SPK, and SPKID

```
Stage2: {
    [keysrc_encryption]bbram_red_key
    [auth_params] ppk_select=1;spk_id=0x12345678
    [ppkfile]keys/primary.pub
    [sskfile]keys/secondary0.pem
    [spksignature]secondary.pub.sha384.sig
    [bootloader,destination_cpu=a53-0, encryption=aes, aeskeyfile=aes0.nky, authentication=rsa] fsbl.elf
    [destination_cpu=a53-0, authentication=rsa] hello_a53_0_64.elf
}
```

The Bootgen command is:

```
bootgen -arch zynqmp -image stage2.bif -o final.bin
```

**Creating a Zynq-7000 SoC Device Boot Image using HSM Mode**

The following figure provides a diagram of an HSM mode boot image for a Zynq®-7000 SoC device. The steps to create this boot image are immediately after the diagram.
Figure 15: Stage 0 to 8 Boot Process

Stage-0

- Generate SPK Hash
- SPK Hash
- SPK

Stage-1

- PSK hash
- Generate PSK signature
- PSK

HSM

Stage-2

- SPK hash
- Generate SPK signature
- SPK

Stage-2a

- Encrypt FSBL
- FSBL

Stage-2b

- SPK signature
- PPK & SPK
- Generate FSBL Hash
- Encrypted FSBL
- FSBL hash

Stage-3

- SSK
- Generate FSBL Signature
- FSBL Signature

Stage-3a

- PPK & SPK
- Generate FSBL Hash
- FSBL hash

Stage-3b

- Insert FSBL Signature
- FSBL Signature

Stage-4

- FSBL hash
- Generate Header Table signature
- Header Table Signature

Stage-4a

- SSK
- Generate FSBL Signature
- FSBL Signature

Stage-4b

- PPK & SPK
- Generate FSBL Hash
- FSBL hash

Stage-5

- PPK & SPK
- Insert FSBL Signature
- FSBL Signature

Stage-5a

- Generate Header Table hash
- Header Table hash

Stage-6

- Headers with Authenticated Certificates
- Generate Boot Image
- Final Boot Image

Stage-7

- Generate Header Table signature
- Header Table Signature

Stage-8

- Generate Boot Image
- Final Boot Image
The process to create a boot image using HSM mode for a Zynq®-7000 SoC device is similar to that of a boot image created using a standard flow with the following BIF file. These examples, where needed, use the OpenSSL program to generate hash files.

```yaml
all:
  
    [aeskeyfile] my_efuse.nky
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [bootloader,encryption=aes,authentication=rsa] zynq_fsbl_0.elf
    [authentication=rsa] system.bit

Stage 0: Generate a hash for SPK

This stage generates the hash of the SPK key.

```yaml
stage0:
  
    [ppkfile] primary.pub
    [spkfile] secondary.pub

The following is the Bootgen command.

```bash
bootgen -image stage0.bif -w -generate_hashes
```

Stage 1: Sign the SPK Hash

This stage creates the signatures by signing the SPK hash

```bash
xil_rsa_sign.exe -gensig -sk primary.pem -data secondary.pub.sha256 -out secondary.pub.sha256.sig
```

Or by using the following OpenSSL program.

```bash
# Swap the bytes in SPK hash
objcopy -I binary -O binary --reverse-bytes=256 secondary.pub.sha256

# Generate SPK signature using OpenSSL
openssl rsautl -raw -sign -inkey primary.pem -in secondary.pub.sha256 > secondary.pub.sha256.sig

# Swap the bytes in SPK signature
objcopy -I binary -O binary --reverse-bytes=256 secondary.pub.sha256.sig
```
Stage 2: Encrypt using AES

This stage encrypts the partition. The `stage2.bif` is as follows.

```plaintext
stage2:
[
    [aeskeyfile] my_efuse.nky
    [bootloader, encryption=aes] zynq_fsbl_0.elf
]
```

The Bootgen command is as follows.

```bash
bootgen -image stage2.bif -w -o fsbl_e.bin -encrypt efuse
```

The output is the encrypted file `fsbl_e.bin`.

Stage 3: Generate Partition Hashes

This stage generates the hashes of different partitions.

Stage 3a: Generate the FSBL Hash

The BIF file is as follows:

```plaintext
stage3a:
[
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature] secondary.pub.sha256.sig
    [bootimage, authentication=rsa] fsbl_e.bin
]
```

The Bootgen command is as follows.

```bash
bootgen -image stage3a.bif -w -generate_hashes
```

The output is the hash file `zynq_fsbl_0.elf.0.sha256`.

Stage 3b: Generate the bitstream hash

The stage3b BIF file is as follows:

```plaintext
stage3b:
[
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature] secondary.pub.sha256.sig
    [authentication=rsa] system.bit
]
```

The Bootgen command is as follows.

```bash
bootgen -image stage3b.bif -w -generate_hashes
```

The output is the hash file `system.bit.0.sha256`.
Stage 4: Sign the Hashes

This stage creates signatures from the partition hash files created.

Stage 4a: Sign the FSBL partition hash

\[ \text{xil_rsa_sign.exe -gensig -sk secondary.pem -data zynq_fsbl_0.elf.0.sha256 -out zynq_fsbl_0.elf.0.sha256.sig} \]

Or by using the following OpenSSL program.

\[
\begin{align*}
\text{#Swap the bytes in FSBL hash} \\
\text{objcopy -I binary -O binary --reverse-bytes=256 zynq_fsbl_0.elf.0.sha256} \\
\text{#Generate FSBL signature using OpenSSL} \\
\text{openssl rsautl -raw -sign -inkey secondary.pem -in zynq_fsbl_0.elf.0.sha256 > zynq_fsbl_0.elf.0.sha256.sig} \\
\text{#Swap the bytes in FSBL signature} \\
\text{objcopy -I binary -O binary --reverse-bytes=256 zynq_fsbl_0.elf.0.sha256.sig}
\end{align*}
\]

The output is the signature file \textit{zynq_fsbl_0.elf.0.sha256.sig}.

Stage 4b: Sign the bitstream hash

\[ \text{xil_rsa_sign.exe -gensig -sk secondary.pem -data system.bit.0.sha256 -out system.bit.0.sha256.sig} \]

Or by using the following OpenSSL program.

\[
\begin{align*}
\text{#Swap the bytes in bitstream hash} \\
\text{objcopy -I binary -O binary --reverse-bytes=256 system.bit.0.sha256} \\
\text{#Generate bitstream signature using OpenSSL} \\
\text{openssl rsautl -raw -sign -inkey secondary.pem -in system.bit.0.sha256 > system.bit.0.sha256.sig} \\
\text{#Swap the bytes in bitstream signature} \\
\text{objcopy -I binary -O binary --reverse-bytes=256 system.bit.0.sha256.sig}
\end{align*}
\]

The output is the signature file \textit{system.bit.0.sha256.sig}.

Stage 5: Insert Partition Signatures

Insert partition signatures created above are changed into authentication certificates.

Stage 5a: Insert the FSBL signature

The \textit{stage5a.bif} BIF is as follows.

\[
\text{stage5a:} \\
\{ \\
\text{[ppkfile] primary.pub} \\
\text{[spkfile] secondary.pub} \\
\text{[spksignature] secondary.pub.sha256.sig} \\
\text{[bootimage, authentication=rsa, presign=zynq_fsbl_0.elf.0.sha256.sig]} \\
\text{fsbl_e.bin} \\
\} 
\]
The Bootgen command is as follows.

```
bootgen -image stage5a.bif -w -o fsbl_e_ac.bin -efuseppkbits efuseppkbits.txt -nonbooting
```

The authenticated output files are `fsbl_e_ac.bin` and `efuseppkbits.txt`.  
**Stage 5b: Insert the bitstream signature**  
The stage5b.bif is as follows.

```
stage5b:
{
  [ppkfile] primary.pub
  [spkfile] secondary.pub
  [spksignature] secondary.pub.sha256.sig
  [authentication=rsa, presign=system.bit.0.sha256.sig] system.bit
}
```

The Bootgen command is as follows.

```
bootgen -image stage5b.bif -o system_e_ac.bin -nonbooting
```

The authenticated output file is `system_e_ac.bin`.

**Stage 6: Generate Header Table Hash**

This stage generates the hash for the header tables.  
The stage6.bif is as follows.

```
stage6:
{
  [bootimage] fsbl_e_ac.bin
  [bootimage] system_e_ac.bin
}
```

The Bootgen command is as follows.

```
bootgen -image stage6.bif -generate_hashes
```

The output hash file is `ImageHeaderTable.sha256`.

**Stage 7: Generate Header Table Signature**

This stage generates the header table signature.

```
xil_rsa_sign.exe -gensig -sk secondary.pem -data ImageHeaderTable.sha256 -out ImageHeaderTable.sha256.sig
```
Or by using the following OpenSSL program:

```bash
# Swap the bytes in header table hash
objcopy -I binary -O binary --reverse-bytes=256 ImageHeaderTable.sha256

# Generate header table signature using OpenSSL
openssl rsautl -raw -sign -inkey secondary.pem -in ImageHeaderTable.sha256
> ImageHeaderTable.sha256.sig

# Swap the bytes in header table signature
objcopy -I binary -O binary --reverse-bytes=256 ImageHeaderTable.sha256.sig
```

The output is the signature file `ImageHeaderTable.sha256.sig`.

**Stage 8: Combine Partitions, Insert Header Table Signature**

The stage8.bif is as follows:

```bash
stage8:
{
  [headersignature] ImageHeaderTable.sha256.sig
  [bootimage] fsbl_e_ac.bin
  [bootimage] system_e_ac.bin
}
```

The Bootgen command is as follows:

```bash
bootgen -image stage8.bif -w -o final.bin
```

The output is the bootimage file `final.bin`.

**Creating a Zynq UltraScale+ MPSoC Device Boot Image using HSM Mode**

The following figure provides a diagram of an HSM mode boot image.
Figure 16: 0 to 10 Stage Boot Process

1. **Stage 0**: Generate SPK Hash
2. **Stage 1**: Generate SPK Signature
3. **Stage 2**: Encrypt FSBL
4. **Stage 2a**: Encrypt FSBL with nky key
5. **Stage 2b**: Generate SPK Hash
6. **Stage 2c**: Generate BH Signature
7. **Stage 3**: Generate BH Hash
8. **Stage 4**: Generate BH Signature
9. **Stage 5**: Generate partition hashes
10. **Stage 6**: Generate partition signatures
11. **Stage 7a**: Encrypt FSBL PPK & SPK with authentication certificate
12. **Stage 7b**: Generate Header Table hash
13. **Stage 7c**: Generate Header Table hash
14. **Stage 8**: Generate Header Table signature
15. **Stage 9**: Generate Header Table signature
16. **Stage 10**: Generate bootmage

Insert FSBL Signature

SPK Signature

BH signature*
To create a boot image using HSM mode for a Zynq® UltraScale+™ MPSoC device, it would be similar to a boot image created using a standard flow with the following BIF file. These examples, where needed, use the OpenSSL program to generate hash files.

```plaintext
all:
{
    [fsbl_config] bh_auth_enable
    [keys_encryption] bbram_red_key
    [pskfile] primary0.pem
    [sskfile] secondary0.pem

    [bootloader,
        destination_cpu=a53-0,
        encryption=aes,
        aeskeyfile=aes0.nky,
        authentication=rsa
    ] fsbl.elf

    [destination_device=pl,
        encryption=aes,
        aeskeyfile=aes1.nky,
        authentication=rsa
    ] system.bit

    [destination_cpu=a53-0,
        authentication=rsa,
        exception_level=el-3,
        trustzone=secure
    ] bl31.elf

    [destination_cpu=a53-0,
        authentication=rsa,
        exception_level=el-2
    ] u-boot.elf
}
```

**Stage 0: Generate a hash for SPK**

The following is the snippet from the BIF file.

```plaintext
stage0:
{
    [ppkfile]primary.pub
    [spkfile]secondary.pub
}
```

The following is the Bootgen command:

```
bootgen -arch zynqmp -image stage0.bif -generate_hashes -w on -log error
```
**Stage 1: Sign the SPK Hash (encrypt the partitions)**

The following is a code snippet using OpenSSL to generate the SPK hash:

```
openssl rsautl -raw -sign -inkey primary0.pem -in secondary.pub.sha384 > secondary.pub.sha384.sig
```

The output of this command is `secondary.pub.sha384.sig`.

**Stage 2a: Encrypt the FSBL**

Encrypt the FSBL using the following snippet in the BIF file.

```
Stage 2a:
{
    [keysrc_encryption] bbram_red_key

    [bootloader.destination_cpu=a53-0,
    encryption=aes,
    aeskeyfile=aes0.nky
    ] fsbl.elf
}
```

The `bootgen` command is:

```
bootgen -arch zynqmp -image stage2a.bif -o fsbl_e.bin -w on -log error
```

**Stage 2b: Encrypt Bitstream**

Generate the following BIF file entry:

```
stage2b:
{
    [encryption=aes,
    aeskeyfile=aes1.nky,
    destination_device=pl,
    pid=1
    ] system.bit
}
```

The `bootgen` command is:

```
bootgen -arch zynqmp -image stage2b.bif -o system_e.bin -w on -log error
```
Stage 3: Generate Boot Header Hash

Generate the boot header hash using the following BIF file:

```bash
stage3:
{
    [fsbl_config] bh_auth_enable
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature] secondary.pub.sha384.sig
    [bootimage,authentication=rsa] fsbl_e.bin
}
```

The Bootgen command is:

```bash
bootgen -arch zynqmp -image stage3.bif -generate_hashes -w on -log error
```

Stage 4: Sign Boot Header Hash

Generate the boot header hash with the following OpenSSL command:

```bash
openssl rsautl -raw -sign -inkey secondary0.pem -in bootheader.sha384 > bootheader.sha384.sig
```

Stage 5: Get Partition Hashes

Get partition hashes using the following command in a BIF file:

```bash
stage5:
{
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature] secondary.pub.sha384.sig
    [bhsignature] bootheader.sha384.sig
    [bootimage,authentication=rsa] fsbl_e.bin
    [bootimage,authentication=rsa] system_e.bin

    [destination_cpu=a53-0, authentication=rsa, exception_level=el-3, trustzone=secure] bl31.elf

    [destination_cpu=a53-0, authentication=rsa, exception_level=el-2] u-boot.elf
}
```

The Bootgen command is:

```bash
bootgen -arch zynqmp -image stage5.bif -generate_hashes -w on -log error
```
Multiple hashes will be generated for a bitstream partition. For more details, see Bitstream Authentication Using External Memory.

The Boot Header hash is also generated from in this stage 5; which is different from the one generated in stage3, because the parameter `bh_auth_enable` is not used in stage5. This can be added in stage5 if needed, but does not have a significant impact because the Boot Header hash generated using stage3 is signed in stage4 and this signature will only be used in the HSM mode flow.

**Stage 6: Sign Partition Hashes**

Create the following files using OpenSSL:

```bash
openssl rsautl -raw -sign -inkey secondary0.pem -in fsbl.elf.0.sha384 > fsbl.elf.0.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in system.bit.0.sha384 > system.bit.0.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in system.bit.1.sha384 > system.bit.1.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in system.bit.2.sha384 > system.bit.2.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in system.bit.3.sha384 > system.bit.3.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in u-boot.elf.0.sha384 > u-boot.elf.0.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in bl31.elf.0.sha384 > bl31.elf.0.sha384.sig
openssl rsautl -raw -sign -inkey secondary0.pem -in bl31.elf.1.sha384 > bl31.elf.1.sha384.sig
```

**Stage 7: Insert Partition Signatures into Authentication Certificate**

**Stage 7a:** Insert the FSBL signature by adding this code to a BIF file:

```bash
Stage7a:
[
    [fsbl_config] bh_auth_enable
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature]secondary.pub.sha384.sig
    [bhsignature]bootheader.sha384.sig
    [bootimage,authentication=rsa,presign=fsbl.elf.0.sha384.sig]fsbl_e.bin
]
```

The Bootgen command is as follows:

```bash
bootgen -arch zynqmp -image stage7a.bif -o fsbl_e_ac.bin -efuseppkbts efuseppkbts.txt -nonbooting -w on -log error
```
Stage 7b: Insert the bitstream signature by adding the following to the BIF file:

```plaintext
stage7b:
{
  [ppkfile]primary.pub
  [spkfile]secondary.pub
  [spksignature]secondary.pub.sha384.sig
  [bhsignature]bootheader.sha384.sig
  [bootimage, authentication=rsa, presign=system.bit.0.sha384.sig]
  system_e.bin
}
```

The Bootgen command is:

```
bootgen -arch zynqmp -image stage7b.bif -o system_e_ac.bin -nonbooting -w on -log error
```

Stage 7c: Insert the U-Boot signature by adding the following to the BIF file:

```plaintext
stage7c:
{
  [ppkfile] primary.pub
  [spkfile] secondary.pub
  [spksignature]secondary.pub.sha384.sig
  [bhsignature]bootheader.sha384.sig
  [destination_cpu=a53-0, authentication=rsa, exception_level=el-2, presign=u-boot.elf.0.sha384.sig]
  u-boot.elf
}
```

The Bootgen command is:

```
bootgen -arch zynqmp -image stage7c.bif -o u-boot_ac.bin -nonbooting -w on -log error
```

Stage 7d: Insert the ATF signature by entering the following into a BIF file:

```plaintext
stage7d:
{
  [ppkfile] primary.pub
  [spkfile] secondary.pub
  [spksignature]secondary.pub.sha384.sig
  [bhsignature]bootheader.sha384.sig
  [destination_cpu=a53-0, authentication=rsa, exception_level=el-3, trustzone=secure, presign=bl31.elf.0.sha384.sig]
  bl31.elf
}
```
The Bootgen command is:

```
bootgen -arch zynqmp -image stage7d.bif -o bl31_ac.bin -nonbooting -w on -log error
```

**Stage 8: Combine Partitions, Get Header Table Hash**

Enter the following in a BIF file:

```
stage8:
{
    [bootimage]fsbl_e_ac.bin
    [bootimage]system_e_ac.bin
    [bootimage]bl31_ac.bin
    [bootimage]u-boot_ac.bin
}
```

The Bootgen command is:

```
bootgen -arch zynqmp -image stage8.bif -generate_hashes -o stage8.bin -w on -log error
```

**Stage 9: Sign Header Table Hash**

Generate the following files using OpenSSL:

```
openssl rsautl -raw -sign -inkey secondary0.pem -in ImageHeaderTable.sha384 > ImageHeaderTable.sha384.sig
```

**Stage 10: Combine Partitions, Insert Header Table Signature**

Enter the following in a BIF file:

```
stage10:
{
    [headersignature]ImageHeaderTable.sha384.sig
    [bootimage]fsbl_e_ac.bin
    [bootimage]system_e_ac.bin
    [bootimage]bl31_ac.bin
    [bootimage]u-boot_ac.bin
}
```

The Bootgen command is:

```
bootgen -arch zynqmp -image stage10.bif -o final.bin -w on -log error
```
FPGA Support

As described in the Chapter 5: Boot Time Security, FPGA-only devices also need to maintain security while deploying them in the field. Xilinx® tools provide embedded IP modules to achieve the Encryption and Authentication, is part of programming logic. Bootgen extends the secure image creation (Encrypted and/or Authenticated) support for FPGA family devices from 7 series and beyond. This chapter details some of the examples of how Bootgen can be used to encrypt and authenticate a bitstream. Bootgen support for FPGAs is available in the standalone Bootgen install.

**Note:** Only bitstreams from 7 series devices and beyond are supported.

Encryption and Authentication

Xilinx® FPGAs use the embedded, PL-based, hash-based message authentication code (HMAC) and an advanced encryption standard (AES) module with a cipher block chaining (CBC) mode.

**Encryption Example**

To create an encrypted bitstream, the AES key file is specified in the BIF using the attribute `aeskeyfile`. The attribute `encryption=aes` should be specified against the bitstream listed in the BIF file that needs to be encrypted.

```
bootgen -arch fpga -image secure.bif -w -o secure.top.bit
```

The BIF file looks like the following:

```
the_ROM_image:
{
   [aeskeyfile] encrypt.nky
   [encryption=aes] top.bit
}
```

**Authentication Example**

A Bootgen command to authenticate an FPGA bitstream is as follows:

```
bootgen -arch fpga -image all.bif -o rsa.bit -w on -log error
```
The BIF file is as follows:

```plaintext
the_ROM_image:
[
  [sskfile] rsaPrivKeyInfo.pem
  [authentication=rsa] plain.bit
]
```

**Family or Obfuscated Key**

To support obfuscated key encryption, you must register with Xilinx support and request the family key file for the target device family. The path to where this file is stored must be passed as a `bif` option before attempting obfuscated encryption. Contact secure.solutions@xilinx.com to obtain the Family Key.

```plaintext
image:
[
  [aeskeyfile] key_file.nky
  [familykey] familyKey.cfg
  [encryption=aes] top.bit
]
```

A sample `aeskey` file is shown in the following image.

![Figure 17: AES Key Sample](image)

**HSM Mode**

For production, FPGAs use the HSM mode, and can also be used in Standard mode.

**Standard Mode**

Standard mode generates a bitstream which has the authentication signature embedded. In this mode, the secret keys are supposed to be available to the user for generating the authenticated bitstream. Run `Bootgen` as follows:

```bash
bootgen -arch fpga -image all.bif -o rsa_ref.bit -w on -log error
```
The following steps listed below describe how to generate an authenticated bitstream in HSM mode, where the secret keys are maintained by secure team and not available with the user. The following figure shows the HSM mode flow:

**Figure 18: HSM Mode Flow**

**Stage 0: Authenticate with dummy key**

This is a one time task for a given bit stream. For stage 0, Bootgen generates the `stage0.bif` file.

```plaintext
the_ROM_image:
[
    [sskfile] dummykey.pem
    [authentication=rsa] plain.bit
]
```

**Note:** The authenticated bitstream has a header, an actual bitstream, a signature and a footer. This `dummy.bit` is created to get a bitstream in the format of authenticated bitstream, with a dummy signature. Now, when the dummy bit file is given to Bootgen, it calculates the signature and inserts at the offset to give an authenticated bitstream.

**Stage 1: Generate hashes**

```
bootgen -arch fpga
    -image stage1.bif -generate_hashes -log error
```
Stage 1. bif is as follows:

```plaintext
the_ROM_image:
{
  [authentication=rsa] dummy.bit
}
```

Stage 2: Sign the Hash HSM, here OpenSSL is used for Demonstration

```plaintext
openssl rsautl -sign
    -inkey rsaPrivKeyInfo.pem -in dummy.sha384 > dummy.sha384.sig
```

Stage 3: Update the RSA certificate with Actual Signature

The Stage3. bif is as follows:

```plaintext
bootgen -arch fpga -image stage3.bif -w -o rsa_rel.bit -log error
```

```plaintext
the_ROM_image:
{
  [spkfile] rsaPubKeyInfo.pem
  [authentication=rsa, presign=dummy.sha384.sig]dummy.bit
}
```

Note: The public key digest, which must be burnt into eFUSEs, can be found in the generated rsaPubKeyInfo.pem.nky file in Stage3 of HSM mode.
Use Cases and Examples

The following are typical use cases and examples for Bootgen. Some use cases are more complex and require explicit instruction. These typical use cases and examples have more definition when you reference the Attributes.

Zynq MPSoC Use Cases

Simple Application Boot on Different Cores

The following example shows how to create a boot image with applications running on different cores. The pmu_fw.elf is loaded by BootROM. The fsbl_a53.elf is the bootloader and loaded on to A53-0 core. The app_a53.elf is executed by A53-1 core, and app_r5.elf by r5-0 core.

```
the_ROM_image:
{
    [pmufw_image] pmu_fw.elf
    [bootloader, destination_cpu=a53-0] fsbl_a53.elf
    [destination_cpu=a53-1] app_a53.elf
    [destination_cpu=r5-0] app_r5.elf
}
```

PMUFW Load by BootROM

This example shows how to create a boot image with pmu_fw.elf loaded by BootROM.

```
the_ROM_image:
{
    [pmufw_image] pmu_fw.elf
    [bootloader, destination_cpu=a53-0] fsbl_a53.elf
    [destination_cpu=r5-0] app_r5.elf
}
```
PMUFW Load by FSBL

This example shows how to create a boot image with `pmu_fw.elf` loaded by FSBL.

```plaintext
the_ROM_image:
{
    [bootloader, destination_cpu=a53-0] fsbl_a53.elf
    [destination_cpu=pmu] pmu_fw.elf
    [destination_cpu=r5-0] app_r5.elf
}
```

**Note:** Bootgen uses the options provided to `[bootloader]` for `[pmufw_image]` as well. The `[pmufw_image]` does not take any extra parameters.

Booting Linux

This example shows how to boot Linux on a Zynq® UltraScale+™ MPSoC device (`arch=zynqmp`).

- The `fsbl_a53.elf` is the bootloader and runs on a53-0.
- The `pmu_fw.elf` is loaded by FSBL.
- The `bl31.elf` is the Arm® Trusted Firmware (ATF), which runs at el-3.
- The U-Boot program, `uboot`, runs at el-2 on a53-0.
- The Linux image, `image.ub`, is placed at offset 0x1E40000 and loaded at 0x10000000.

```plaintext
the_ROM_image:
{
    [bootloader, destination_cpu = a53-0]fsbl_a53.elf
    [destination_cpu=pmu]pmu_fw.elf
    [destination_cpu=a53-0, exception_level=el-3, trustzone]bl31.elf
    [destination_cpu=a53-0, exception_level=el-2] u-boot.elf
    [offset=0x1E40000, load=0X10000000, destination_cpu=a53-0]image.ub
}
```

Encryption Flow: BBRAM Red Key

This example shows how to create a boot image with the encryption enabled for FSBL and the application with the Red key stored in BBRAM:

```plaintext
the_ROM_image:
{
    [keysrsrc_encryption] bbram_red_key
    [bootloader, encryption=aes, aekyfile=aes0.nky, destination_cpu=a53-0]ZynqMP_Fsbl.elf
    [destination_cpu=a53-0, encryption=aes, aekyfile=aes1.nky]App_A53_0.elf
}
```
Encryption Flow: Red Key Stored in eFUSE

This example shows how to create a boot image with encryption enabled for FSBL and application with the RED key stored in eFUSE.

```plaintext
the_ROM_image:
[
  [keysrc_encryption] efuse_red_key
  [
    bootloader,
    encryption=aes,
    aeskeyfile=aes0.nky,
    destination_cpu=a53-0
  ] ZynqMP_Fsbl.elf
  [
    destination_cpu = a53-0,
    encryption=aes,
    aeskeyfile=aes1.nky
  ] App_A53_0.elf
]
```

Encryption Flow: Black Key Stored in eFUSE

This example shows how to create a boot image with the encryption enabled for FSBL and an application with the efuse_blk_key stored in eFUSE. Authentication is also enabled for FSBL.

```plaintext
the_ROM_image:
{
  [fsbl_config] puf4kmode, shutter=0x01000010
  [auth_params] ppk_select=0; spk_id=0x584C4E58
  [pskfile] primary_4096.pem
  [sskfile] secondary_4096.pem
  [keysrc_encryption] efuse_blk_key
  [bh_key_iv] bhkeyiv.txt
  [
    bootloader,
    encryption=aes,
    aeskeyfile=aes0.nky,
    authentication=rsa
  ] fsbl.elf
}
```

**Note:** Boot image authentication is compulsory for using black key encryption.
Encryption Flow: Black Key Stored in Boot Header

This example shows how to create a boot image with encryption enabled for FSBL and the application with the `bh_blk_key` stored in the Boot Header. Authentication is also enabled for FSBL.

```plaintext
the_ROM_image:
{
    [pskfile] PSK.pem
    [sskfile] SSK.pem
    [fsbl_config] shutter=0x0100005E
    [auth_params] ppk_select=0
    [bh_keyfile] blackkey.txt
    [bh_key_iv] black_key_iv.txt
    [puf_file] helperdata4k.txt
    [keysrch_encryption] bh_blk_key
    {
        bootloader,
        encryption=aes,
        aeskeyfile=aes0.nky,
        authentication=rsa,
        destination_cpu=a53-0
    } ZynqMP_Fsbl.elf

    [destination_cpu=a53-0,
    encryption=aes,
    aeskeyfile=aes1.nky
    ] App_A53_0.elf
}

Note: Boot image Authentication is required when using black key Encryption.

Encryption Flow: Gray Key Stored in eFUSE

This example shows how to create a boot image with encryption enabled for FSBL and the application with the `efuse_gry_key` stored in eFUSE.

```plaintext
the_ROM_image:
{
    [keysrch_encryption] efuse_gry_key
    [bh_key_iv] bh_key_iv.txt
    {
        bootloader,
        encryption=aes,
        aeskeyfile=aes0.nky,
        destination_cpu=a53-0
    } ZynqMP_Fsbl.elf

    [destination_cpu=a53-0,
    encryption=aes,
    aeskeyfile=aes1.nky
    ] App_A53_0.elf
}
Encryption Flow: Gray Key stored in Boot Header

This example shows how to create a boot image with encryption enabled for FSBL and the application with the \texttt{bh\_gry\_key} stored in the Boot Header.

\begin{verbatim}
the_ROM_image:
[
  [keysrc_encryption] bh_gry_key
  [bh_keyfile] bhkey.txt
  [bh_key_iv] bh_key_iv.txt

  [bootloader,
   encryption=aes,
   aeskeyfile=aes0.nky,
   destination_cpu=a53-0
  ] ZynqMP_Fsbl.elf

  [destination_cpu=a53-0,
   encryption=aes,
   aeskeyfile=aes1.nky
  ] App_A53_0.elf
]
\end{verbatim}

Operational Key

This example shows how to create a boot image with encryption enabled for FSBL and the application with the gray key stored in the Boot Header. This example shows how to create a boot image with encryption enabled for FSBL and application with the red key stored in eFUSE.

\begin{verbatim}
the_ROM_image:
[
  [fsbl_config] opt_key
  [keysrc_encryption] efuse_red_key

  [bootloader,
   encryption=aes,
   aeskeyfile=aes0.nky,
   destination_cpu=a53-0
  ] ZynqMP_Fsbl.elf

  [destination_cpu=a53-0,
   encryption=aes,
   aeskeyfile=aes1.nky
  ] App_A53_0.elf
]
\end{verbatim}
Using Op Key to Protect the Device Key in a Development Environment

The following steps provide a solution in a scenario where two development teams, Team-A (secure team), which manages the secret red key and Team-B, (Not so secure team), work collaboratively to build an encrypted image without sharing the secret red key. Team-A manages the secret red key. Team-B builds encrypted images for development and test. However, it does not have access to the secret red key.

Team-A encrypts the boot loader with the device key (using the Op_key option) - delivers the encrypted bootloader to Team-B. Team-B encrypts all the other partitions using the Op_key.

Team-B takes the encrypted partitions that they created, and the encrypted boot loader they received from the Team-A and uses bootgen to stitch everything together into a single boot.bin.

The following procedures describe the steps to build an image:

Procedure-1

In the initial step, Team-A encrypts the boot loader with the device Key using the opt_key option, delivers the encrypted boot loader to Team-B. Now, Team-B can create the complete image at a go with all the partitions and the encrypted boot loader using Operational Key as Device Key.

1. Encrypt Bootloader with device key:

```
bootgen -arch zynqmp -image stage1.bif -o fsbl.e.bin -w on -log error
```

Example stage1.bif:

```
stage1:
{
    [fsbl_config] opt_key
    [keysrcc_encryption] bbram_red_key
    [bootloader,
    destination_cpu=a53-0,
    encryption=aes,aeskeyfile=aes.nky
    ] fsbl.elf
}
```

Example aes.nky for stage1:

```
Device xc7z020clg484;
Key 0 AD00C233E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
IV 0 F7F8FDE8674A28D66ED8E37;
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC964778CBAF457D613508F;
```

2. Attach the encrypted bootloader and rest of the partitions with Operational Key as device Key, to form a complete image:

```
bootgen -arch zynqmp -image stage2a.bif -o final.bin -w on -log error
```
Example of **stage2.bif**:

```plaintext
stage2:
{
    [bootimage]fsbl_e.bin
    [
        destination_cpu=a53-0,
        encryption=aes,
        aeskeyfile=aes-opt.nky
    ] hello.elf
    [
        destination_cpu=a53-1,
        encryption=aes,
        aeskeyfile=aes-opt1.nky
    ] hello1.elf
}
```

**Example aes-opt.nky for stage2:**

```
Device xc7z020clg484:
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F:
IV 0 F7F8FDE08674A28DC6ED8E37:
```

**Procedure-2:** In the initial step, Team-A encrypts the boot loader with the device Key using the `opt_key` option, delivers the encrypted boot loader to Team-B. Now, Team-B can create encrypted images for each partition independently, using the Operational Key as Device Key. Finally, Team-B can use bootgen to stitch all the encrypted partitions and the encrypted boot loader, to get the complete image.

1. **Encrypt Bootloader with device key:**

   ```
   bootgen -arch zynqmp -image stage1.bif -o fsbl_e.bin -w on -log error
   ```

**Example stage1.bif:**

```plaintext
stage1:
{
    [fsbl_config] opt_key
    [keysrc_encryption] bbram_red_key
    [
        bootloader,
        destination_cpu=a53-0,
        encryption=aes,aeskeyfile=aes.nky
    ] fsbl.elf
}
```

**Example aes.nky for stage1:**

```
Device xc7z020clg484:
Key 0 AD00C03E3D238492C819456A98C124AE890ACEF002100128932:
IV 0 F7F8FDE08674A28DC6ED8E37:
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F
```

2. **Encrypt the rest of the partitions with Operational Key as device key:**

   ```
   bootgen -arch zynqmp -image stage2a.bif -o hello_e.bin -w on -log error
   ```
Example of `stage2a.bif`:

```plaintext
stage2a:
{
  [destination_cpu=a53-0,
   encryption=aes,
   aeskeyfile=aes-opt.nky
  ] hello.elf
}
bootgen -arch zynqmp -image stage2b.bif -o hello1_e.bin -w on -log error
```

Example of `stage2b.bif`:

```plaintext
stage2b:
{
  [aeskeyfile] aes-opt.nky
  [
   destination_cpu=a53-1,
   encryption=aes,
   aeskeyfile=aes-opt.nky
  ] hello1.elf
}
```

Example of `aes-opt.nky` for `stage2a` and `stage2b`:

```
Device xc7z020clg484;
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
IV 0 F7F8FDE08674A28EDC6ED8E37;
```

3. Use Bootgen to stitch the above example to form a complete image:

```plaintext
Use bootgen to stitch the above, to form a complete image.
```

Example of `stage3.bif`:

```plaintext
stage3:
{
  [bootimage]fsbl_e.bin
  [bootimage]hello_e.bin
  [bootimage]hello1_e.bin
}
```

Note: `opt_key` of `aes.nky` is same as `Key 0` in `aes-opt.nky` and `IV 0` must be same in both nky files.

**Single Partition Image**

This feature provides support for authentication and/or decryption of single partition (non-bitstream) image created by Bootgen at U-Boot prompt.

Note: This feature does not support images with multiple partitions.

**u-boot command for loading secure images**

```
zyngmp secure <srcaddr> <len> [key_addr]
```
This command verifies secure images of $len\ bytes$ long at address $src$. Optional key_addr can be specified if user key needs to be used for decryption.

**Only Authentication Use Case**

To use only authentication at U-Boot, create the authenticated image using `bif` as shown in the following example.

1. Create a single partition image that is authenticated at U-Boot.
   
   *Note:* If you provide an `elf` file, it should not contain multiple loadable sections. If your `elf` file contains multiple loadable sections, you should convert the input to the `.bin` format and provide the `.bin` as input in `bif`. An example `bif` is as follows:

   ```
   the_ROM_image:
   [pskfile]rsa4096_private1.pem
   [sskfile]rsa4096_private2.pem
   [auth_params] ppk_select=1;spk_id=0x12345678
   [authentication = rsa]Data.bin
   ```

2. When the image is generated, download the authenticated image to the DDR.

3. Execute the U-Boot command to authenticate the secure image as shown in the following example.

   ```
   ZynqMP> zynqmp secure 100000 2d000
   Verified image at 0x102800
   ```

4. U-Boot returns the start address of the actual partition after successful authentication. U-Boot prints and error code in the event of a failure. If RSA_EN eFUSE is programmed, image authentication is mandatory. Boot header authentication is not supported when eFUSE RSA enabled.

**Only Encryption Use Case**

In case the image is only encrypted, there is no support for device key. When authentication is not enabled, only KUP key decryption is supported.

**Authentication Flow**

This example shows how to create a boot image with authentication enabled for FSBL and application with Boot Header authentication enabled to bypass the PPK hash verification:

```
the_ROM_image:
[fsbl_config] bh_auth_enable
[auth_params] ppk_select=0; spk_id=0x00000000
[pskfile] PSK.pem
[sskfile] SSK.pem
```
BIF File with SHA-3 eFUSE RSA Authentication and PPK0

This example shows how to create a boot image with authentication enabled for FSBL and the application with boot header authentication enabled to bypass the PPK hash verification:

```json
the_ROM_image:
{
[auth_params] ppk_select=0; spk_id=0x00000000
[pskfile] PSK.pem
[sskfile] SSK.pem
[
[bootloader,
  authentication=rsa,
  destination_cpu=a53-0
] ZynqMP_Fsbl.elf
 [destination_cpu=a53-0, encryption=aes] App_A53_0.elf
]}
```

XIP

This example shows how to create a boot image that executes in place for a zynqmp (Zynq® UltraScale+™ MPSoC):

```json
the_ROM_image:
{
[
[bootloader,
  destination_cpu=a53-0,
  xip_mode
] mpsoc_qspi_xip.elf
]
```

See xip_mode for more information about the command.
BIF Attribute Reference

aarch32_mode

Syntax

```
[aarch32_mode] <partition>
```

Description

To specify the binary file is to be executed in 32-bit mode.

**Note:** Bootgen automatically detects the execution mode of the processors from the `.elf` files. This is valid only for binary files.

Arguments

Specified partition.

Example

```
the_ROM_image:
[
    [bootloader, destination_cpu=a53-0] zynqmp_fsbl.elf
    [destination_cpu=a53-0, aarch32_mode] hello.bin
    [destination_cpu=r5-0] hello_world.elf
]
```

aeskeyfile

Syntax

```
[aeskeyfile = <keyfile name>] <partition>
[aeskeyfile] <key filename>
```
Description
The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file must be written to eFUSE or BBRAM. If the key file is not present in the path specified, a new key is generated by Bootgen, which is used for encryption.

Note: For Zynq® UltraScale+™ MPSoC only: Multiple key files need to be specified in the BIF file. Key0, IV0 and Key Opt should be the same across all nky files that will be used. For cases where multiple partitions are generated for an ELF file, each partition can be encrypted using keys from a unique key file. Refer to the following examples.

Arguments
Specified file name.

Return Value
None

Zynq-7000 SoC Example
The partitions fsbl.elf and hello.elf are encrypted using keys in test.nky.

```
all:
{
  [keys src_encryption] bbram_red_key
  [aeskeyfile] test.nky
  [bootloader, encryption=aes] fsbl.elf
  [encryption=aes] hello.elf
}
```

Sample key (.nky) file - test.nky

```
Device       xc7z020clg484;
Key 0        8177B12032A7DEEE35D0F71A7FC399027BF...D608C58;
Key StartCBC 952FD2DF1DA543C46CDDE4F811506228;
Key HMAC     123177B12032A7DEEE35D0F71A7FC3990BF...127BD89;
```

Zynq UltraScale+ MPSoC Example

Example 1:
The partition fsbl.elf is encrypted with keys in test.nky, hello.elf using keys in test1.nky and app.elf using keys in test2.nky. Sample BIF - test_multipl.bif.

```
all:
{
  [keys src_encryption] bbram_red_key
  [bootloader, encryption=aes, aeskeyfile=test.nky] fsbl.elf
  [encryption=aes, aeskeyfile=test1.nky] hello.elf
  [encryption=aes, aeskeyfile=test2.nky] app.elf
}
```
Example 2:
Consider Bootgen creates three partitions for hello.elf, called hello.elf.0, hello.elf.1, and hello.elf.2. Sample BIF - test_multiple.bif

```bash
all:
{
    [keysrc_encryption] bbram_red_key
    [bootloader,encryption=aes,aeskeyfile=test.nky] fsbl.elf
    [encryption=aes,aeskeyfile=test1.nky] hello.elf
}
```

Additional information:

- The partition fsbl.elf is encrypted with keys in test.nky. All hello.elf partitions are encrypted using keys in test1.nky.
- You can have unique key files for each hello partition by having key files named test1.1.nky and test1.2.nky in the same path as test1.nky.
- hello.elf.0 uses test1.nky
- hello.elf.1 uses test1.1.nky
- hello.elf.2 uses test1.2.nky
- If any of the key files (test1.1.nky or test1.2.nky) is not present, Bootgen generates the key file.

---

**alignment**

**Syntax**

```
[alignment= <value>] <partition>
```

Sets the byte alignment. The partition will be padded to be aligned to a multiple of this value. This attribute cannot be used with offset.

**Arguments**

Number of bytes to be aligned.

**Example**

```bash
all:
{
    [bootloader] fsbl.elf
    [alignment=64] u-boot.elf
}
```
auth_params

- Syntax:

```
[auth_params] ppk_select=<0|1>; spk_id <32-bit spk id>:
spk_select=<spk-efuse/user-efuse>; auth_header
```

Description

Authentication parameters specify additional configuration such as which PPK, SPK to use for authentication of the partitions in the boot image. Arguments for this bif parameter are:

- ppk_select: Selects which PPK to use. Options are 0 (default) or 1.
- spk_id: Specifies which SPK can be used or revoked. See User eFUSE Support with Enhanced RSA Key Revocation. The default value is 0x00.
- spk_select: To differentiate spk and user efuses. Options are spk-efuse (default) and user_efuse.
- header_auth: To authenticate headers when no partition is authenticated.

Note:

1. ppk_select is unique for each image.
2. Each partition can have its own spk_select and spk_id.
3. spk-efuse id is unique across the image, but user-efuse id can vary between partitions.
4. spk_select/spk_id outside the partition scope will be used for headers and any other partition that does not have these specifications as partition attributes.

Example

Sample BIF 1 - test.bif

```plaintext
all:
[
    [auth_params] ppk_select=0; spk_id=0x12345678
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [bootloader, authentication=rsa] fsbl.elf
]
```
Sample BIF 2 - test.bif

```bash
call:
[
    [auth_params] ppk_select=0;spk_select=user-efuse;spk_id=0x22
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [bootloader, authentication = rsa]
    fsbl.elf
]
```

Sample BIF 3 - test.bif

```bash
call:
[
    [auth_params] ppk_select=1; spk_select=user-efuse; spk_id=0x22;
    header_auth
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [destination_cpu=a53-0] test.elf
]
```

Sample BIF 4 - test.bif

```bash
call:
[
    [auth_params] ppk_select=1;spk_select=user-efuse;spk_id=0x22
    [pskfile] primary.pem
    [sskfile] secondary0.pem

    /* FSBL - Partition-0) */
    [bootloader,
    destination_cpu = a53-0,
    authentication = rsa,
    spk_id = 0x12345678,
    spk_select = spk-efuse,
    sskfile = secondary1.pem
    ] fsb53a53.elf

    /* Partition-1 */
    [destination_cpu = a53-1,
    authentication = rsa,
    spk_id = 0x24,
    spk_select = user-efuse,
    sskfile = secondary2.pem
    ] hello.elf
]
```
**authentication**

**Syntax**

```
[authentication=<option>] <partition>
```

**Description**

This specifies the partition to be authenticated.

**Arguments**

- **none**: Partition not authenticated. This is the default value.
- **rsa**: Partition authenticated using RSA algorithm.

**Example**

Sample BIF - test.bif

```diff
all:
[
  [ppkfile] ppk.txt
  [spkfile] spk.txt
  [bootloader,authentication=rsa] fsbl.elf
  [authentication=rsa] hello.elf
]
```

---

**big_endian**

**Syntax**

```
[big_endian] <partition>
```

**Description**

To specify the binary file is in big endian format.

**Note**: Bootgen automatically detects the endianness of .elf files. This is valid only for binary files.

**Arguments**

Specified partition.
Example

```
the_ROM_image:
{
  [bootloader, destination_cpu=a53-0] zynqmp_fsbl.elf
  [destination_cpu=a53-0, big_endian] hello.bin
  [destination_cpu=r5-0] hello_world.elf
}
```

---

**bh_keyfile**

**Syntax**

```
[bh_keyfile] <key file path>
```

**Description**

256-bit obfuscated key or black key to be stored in boot header. This is only valid when the encryption key source is either grey key or black key.

**Arguments**

Path to the obfuscated key or black key, based on which source is selected.

**Example**

Sample BIF - test.bif

```
all:
{
  [keysrc_encryption] bh_gry_key
  [bh_keyfile] obfuscated_key.txt
  [bh_key_iv] obfuscated_iv.txt
  [bootloader, encryption=aes, aeskeyfile=encr.nky, destination_cpu=a53-0]fsbl.elf
}
```

---

**bh_key_iv**

**Syntax**

```
[bh_key_iv] <iv file path>
```

**Description**

Initialization vector used when decrypting the obfuscated key or black key.
Arguments

Path to file.

Example

Sample BIF - test.bif
all:
{
    [keysrc_encryption] bh_gry_key
    [bh_keyfile] obfuscated_key.txt
    [bh_key_iv] obfuscated_iv.txt
    [bootloader, encryption=aes, aeskeyfile=encr.nky, destination_cpu=a53-0] fsbl.elf
}

bhsignature

Syntax

[bhsignature] <signature-file>

Description

Imports Boot Header signature into authentication certificate. This can be used if you do not want to share the secret key PSK. You can create a signature and provide it to Bootgen.

Example

all:
{
    [ppkfile] ppk.txt
    [spkfile] spk.txt
    [spksignature] spk.txt.sha384.sig
    [bhsignature] bootheader.sha384.sig
    [bootloader, authentication=rsa] fsbl.elf
}

blocks

Syntax

[blocks = <size><num>;<size><num>;...;<size><*>] <partition>
Description

Specify block sizes for key-rolling feature in encryption. Each module is encrypted using its own unique key. The initial key is stored at the key source on the device, while keys for each successive module are encrypted (wrapped) in the previous module.

Arguments

The <size> mentioned is taken in Bytes. If the size is specified as X(*), then all the remaining blocks will be of the size ‘X’.

Example

Sample BIF - test.bif
all:
  { [keystore_encryption] bbram_red_key
    [bootloader,encryption=aes, aeskeyfile=encr.nky,
     destination_cpu=a53-0,blocks=4096(2):1024:2048(2):4096(*)]
    fsbl.elf
  }

Note: In the above example, the first two blocks are of 4096 bytes, the second block is of 1024 bytes, and the next two blocks are of 2048 bytes. The rest of the blocks are of 4096 bytes.

---

boot_device

Syntax

[boot_device] <options>

Description

Specifies the secondary boot device. Indicates the device on which the partition is present.

Arguments

Options are:

- qspi32
- qspi24
- nand
- sd0
- sd1
• sd-1s
• mmc
• usb
• ethernet
• pcie
• sata

Example

```all:
  [boot_device]sd0
  [bootloader,destination_cpu=a53-0]fsbl.elf
]```

---

**bootimage**

**Syntax**

```[bootimage] <image created by bootgen>```

**Description**

This specifies that the following file specification is a bootimage that was created by Bootgen, being reused as input.

**Arguments**

Specified file name.

**Example**

```all:
  [bootimage]fsbl.bin
  [bootimage]system.bin
]```

In the above example, the `fsbl.bin` and `system.bin` are images generated using Bootgen.
Example fsbl.bin generation

```
image:
{
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [bootloader, authentication=rsa, aeskeyfile=encr_key.nky, encryption=aes] fsbl.elf
}
```

Command: bootgen -image fsbl.bif -o fsbl.bin -encrypt efuse

Example system.bin generation

```
image:
{
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [authentication=rsa] system.bit
}
```

Command: bootgen -image system.bif -o system.bin

---

**bootloader**

**Syntax**

```
[bootloader] <partition>
```

**Description**

Identifies an ELF file as the FSBL.

- Only ELF files can have this attribute.
- Only one file can be designated as the bootloader.
- The program header of this ELF file must have only one LOAD section with filesz >0, and this section must be executable (x flag must be set).

**Arguments**

Specified file name.
Example

```plaintext
all:
  [bootloader] fsbl.elf
  hello.elf
```

### bootvectors

**Syntax**

```
[bootvectors] <values>
```

**Description**

This attribute specifies the vector table for eXecute in Place (XIP).

**Example**

```plaintext
all:
  {
    [bootvectors]0x14000000,0x14000000,0x14000000,0x14000000,0x14000000,0x14000000,0x14000000,0x14000000
    [bootloader,destination_cpu=a53-0]fsbl.elf
  }
```

### checksum

**Syntax**

```
[checksum = <options>] <partition>
```

**Description**

This specifies the partition needs to be checksummed. This is not supported along with more secure features like authentication and encryption.

**Arguments**

- none: No checksum operation.
- MD5: MD5 checksum operation for Zynq®-7000 SoC devices. In these devices, checksum operations are not supported for bootloaders.
destination_cpu

Syntax

[destination_cpu <options>] <partition>

Description

Specifies which core will execute the partition. The following example specifies that FSBL will be executed on A53-0 core and application on R5-0 core.

Note:

- FSBL can only run on either A53-0 or R5-0.
- PMU loaded by FSBL: [destination_cpu=pmu] pmu.elf In this flow, BootROM loads FSBL first, and then FSBL loads the PMU firmware.
- PMU loaded by BootROM: [pmufw_image] pmu.elf. In this flow, BootROM loads PMU first and then the FSBL so PMU does the power management tasks, before the FSBL comes up.

Arguments

- a53-0 (default)
- a53-1
- a53-2
- a53-3
- r5-0
- r5-1
- r5-lockstep
- pmu

Example

all:
[
   [ bootloader, destination_cpu=a53-0] fsbl.elf
   [ destination_cpu=r5-0] app.elf
]
destination_device

Syntax

```plaintext
[destination_device <options>] <partition>
```

Description

Specifies whether the partition is targeted for PS or PL.

Arguments

- `ps`: The partition is targeted for PS. This is the default value.
- `pl`: The partition is targeted for PL, for bitstreams.

Example

```plaintext
all:
{
    [bootsloader,destination_cpu=a53-0]fsbl.elf
    [destination_device=pl]system.bit
    [destination_cpu=r5-1]app.elf
}
```

early_handoff

Syntax

```plaintext
[early_handoff] <partition>
```

Description

This flag ensures that the handoff to applications that are critical immediately after the partition is loaded; otherwise, all the partitions are loaded sequentially and handoff also happens in a sequential fashion.

**Note**: In the following scenario, the FSBL loads app1, then app2, and immediately hands off the control to app2 before app1.
**encryption**

**Syntax**

```
[encryption = <options>] <partition>
```

**Description**

This specifies the partition needs to be encrypted. Encryption Algorithms are:

**Arguments**

- `none`: Partition not encrypted. This is the default value.
- `aes`: Partition encrypted using AES algorithm.

**Example**

```
all:
{
  [aeskeyfile]test.nky
  [bootloader, encryption=aes] fsbl.elf
  hello.elf
}
```
Arguments

- el-0
- el-1
- el-2
- el-3 (default)

Example

```
all:
{
[bootloader, destination_cpu=a53-0] fsbl.elf
[destination_cpu=a53-0, exception_level=el-3] bl31.elf
[destination_cpu=a53-0, exception_level=el-2] u-boot.elf
}
```

familykey

Syntax

```
[familykey] <key file path>
```

Description

Specify Family Key. To obtain family key, contact a Xilinx® representative at secure.solutions@xilinx.com.

Arguments

Path to file.

Example

```
all:
{
[aeskeyfile] encr.nky
[bh_key_iv] bh_iv.txt
[familykey] familykey.cfg
}
```
fsbl_config

Syntax

[fsbl_config <options>] <partition>

Description

This option specifies the parameters used to configure the boot image. FSBL, which should run on A53 in 64-bit mode in Boot Header authentication mode.

Arguments

- bh_auth_enable: Boot Header Authentication Enable: RSA authentication of the bootimage will be done excluding the verification of PPK hash and SPK ID.
- auth_only: Boot image is only RSA signed. FSBL should not be decrypted.
- opt_key: Optional key is used for block-0 decryption. Secure Header has the opt key.
- pufhd_bh: PUF helper data is stored in Boot Header. (Default is efuse) PUF helper data file is passed to bootgen using the [puf_file] option.
- puf4kmode: PUF is tuned to use in 4k bit configuration. (Default is 12k bit). shutter = <value> 32 bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time.

Example

all:
{
  [fsbl_config] bh_auth_enable
  [pskfile] primary.pem
  [sskfile] secondary.pem
  [bootloader,destination_cpu=a53-0,authentication=rsa] fsbl.elf
}

headersignature

Syntax

[headersignature] <signature file>
Description

Imports the Header signature into the Authentication Certificate. This can be used in case the user does not want to share the secret key. The user can create a signature and provide it to Bootgen.

Arguments

<signature_file>

Example

```yaml
all:
  [ppkfile] ppk.txt
  [spkfile] spk.txt
  [headersignature] headers.sha256.sig
  [spksignature] spk.txt.sha256.sig
  [bootloader, authentication=rsa] fsbl.elf
```

hivec

Syntax

```
[hivec] <partition>
```

Description

To specify the location of Exception Vector Table as hivec. This is applicable with a53 (32 bit) and r5 cores only.

- hivec: exception vector table at 0xFFFF0000.
- lovec: exception vector table at 0x00000000. This is the default value.

Arguments

None
Example
A sample BIF file is shown below:

```
all:
  [bootloader, destination_cpu=a53_0]fsbl.elf
  [destination_cpu=r5-0,hivec]app1.elf
```

**init**

**Syntax**

```
[init] <filename>
```

**Description**

Register initialization block at the end of the bootloader, built by parsing the .int file specification. Maximum of 256 address-value init pairs are allowed. The .int files have a specific format.

**Example**

A sample BIF file is shown below:

```
all:
  [init] test.int
```

**keysrc_encryption**

**Syntax**

```
[keysrc_encryption] <options> <partition>
```

**Description**

This specifies the Key source for encryption.

**Arguments**

- `bbram_red_key`: RED key stored in BBRAM
• **efuse_red_key**: RED key stored in efuse
• **efuse_gry_key**: Grey (Obfuscated) Key stored in eFUSE.
• **bh_gry_key**: Grey (Obfuscated) Key stored in boot header.
• **bh_blk_key**: Black Key stored in boot header.
• **efuse_blk_key**: Black Key stored in eFUSE.
• **kup_key**: User Key.

**Example**

```plaintext
all:
[ keysrc_encryption] efuse_gry_key
  [bootloader, encryption=aes, aeskeyfile=encr.nky, destination_cpu=a53-0] fsbl.elf
```

FSBL is encrypted using the key *encr.nky*, which is stored in the efuse for decryption purpose.

---

**load**

**Syntax**

```
[load=<value>] <partition>
```

**Description**

Sets the load address for the partition in memory.

**Example**

```plaintext
all:
[ 
  [bootloader] fsbl.elf
  u-boot.elf
  [load=0x3000000, offset=0x500000] uImage.bin
  [load=0x2A00000, offset=0xa00000] devicetree.dtb
  [load=0x2000000, offset=0xc00000] uramdisk.image.gz
]
```
offset

Syntax

```
[offset=<value>] <partition>
```

Description

Sets the absolute offset of the partition in the boot image.

Arguments

Specified value and partition.

Example

```
all:
{
    [bootloader] fsbl.elf u-boot.elf
    [load=0x3000000, offset=0x500000] uImage.bin
    [load=0x2A00000, offset=0xa00000] devicetree.dtb
    [load=0x2000000, offset=0xc00000] uramdisk.image.gz
}
```

partition_owner

Syntax

```
[partition_owner = <options>] <partition>
```

Description

Owner of the partition which is responsible to load the partition.

Arguments

- fsbl (default)
- u-boot
Example

```
all:
[
    [bootloader]fsbl.elf
    [partition_owner=uboot] hello.elf
]
```

### pid

**Syntax**

```
[pid = <id_no>] <partition>
```

**Description**

This specifies the partition id. The default value is 0.

**Example**

```
all:
[
    [encryption=aes, aeskeyfile=test.nky, pid=1] hello.elf
]
```

### pmufw_image

**Syntax**

```
[pmufw_image] <PMU ELF file>
```

**Description**

PMU Firmware image to be loaded by BootROM, before loading the FSBL. The options for the pmufw_image are inline with the bootloader partition. Bootgen does not consider any extra attributes given along with the pmufw_image option.

**Arguments**

Filename
Example

```
the_ROM_image:
{
    [pmufw_image] pmu_fw.elf
    [bootloader, destination_cpu=a53-0] fsbl_a53.elf
    [destination_cpu=a53-1] app_a53.elf
    [destination_cpu=r5-0] app_r5.elf
}
```

ppkfile

Syntax

```
[ppkfile] <key filename>
```

Description

The Primary Public Key (PPK) key is used to authenticate partitions in the boot image.

See Using Authentication

Arguments

Specified file name.

Note: The secret key file contains the public key component of the key. You need not specify the public key (PPK) when the secret key (PSK) is mentioned.

Example

```
all:
{
    [ppkfile] primarykey.pub
    [pskfile] primarykey.pem
    [spkfile] secondarykey.pem
    [sskfile] secondarykey.pem
    [bootloader, authentication=rsa] fsbl.elf
    [authentication=rsa] hello.elf
}
```

presign

Syntax

```
[presign = <signature_file>] <partition>
```
**Description**

Imports partition signature into partition authentication certificate. Use this if you do not want to share the secret key (SSK). You can create a signature and provide it to Bootgen.

- `<signature_file>`: Specifies the signature file.
- `<partition>`: Lists the partition to which to apply to the signature_file.

**Example**

```yaml
all:
  [ppkfile] ppk.txt
  [spkfile] spk.txt
  [headsignature] headers.sha256.sig
  [spksignature] spk.txt.sha256.sig
  [bootloader, authentication=rsa, presign=fsbl.sig]fsbl.elf
```

**pskfile**

**Syntax**

```
[pskfile] <key filename>
```

**Description**

This Primary Secret Key (PSK) is used to authenticate partitions in the boot image. For more information, see Using Authentication

**Arguments**

Specified file name.

**Note:** The secret key file contains the public key component of the key. You need not specify the public key (PPK) when the secret key (PSK) is mentioned.

**Example**

```yaml
all:
  [pskfile]primarykey.pem
  [sskfile]secondarykey.pem
  [bootloader, authentication=rsa]fsbl.elf
  [authentication=rsa] hello.elf
```
**puf_file**

**Syntax**

```
[puf_file] <puf data file>
```

**Description**

PUF helper data file.

- PUF is used with black key as encryption key source.
- PUF helper data is of 1544 bytes.
- 1536 bytes of PUF HD + 4 bytes of CHASH + 3 bytes of AUX + 1 byte alignment.

See Black/PUF Keys for more information.

**Example**

```yaml
all:
[
    [fsbl_config] pufhd_bh
    [puf_file] pufhelperdata.txt
    [bh_keyfile] black_key.txt
    [bh_key_iv] bhkeyiv.txt
    [bootloader,destination_cpu=a53-0,encryption=aes] fsbl.elf
]
```

---

**reserve**

**Syntax**

```
[reserve=<value>] <partition>
```

**Description**

Reserves the memory and padded after the partition. The value specified for reserving the memory is in bytes.

**Arguments**

Specified partition
Example

```all:
[bootloader]fsbl.elf
[reserve=0x1000]test.bin
]```

split

Syntax

```[split] mode = <mode-options>, fmt=<format>```  

Description

Splits the image into parts based on mode. Slaveboot mode splits as follows:

- Boot Header + Bootloader
- Image and Partition Headers
- Rest of the partitions

Normal mode splits as follows:

- Bootheader + Image Headers + Partition Headers + Bootloader
- Partition1
- Partition2 and so on

Slaveboot is supported only for ZynqMP, normal is supported for both Zynq and ZynqMP. Along with the split mode, output format can also be specified as `bin` or `mcs`.

Options

The available options for argument mode are:

- slaveboot
- normal
- bin
- mcs
Example

```
all:
[
  [split]mode=slaveboot,fmt=bin
  [bootloader,destination_cpu=a53-0]fsbl.elf
  [destination_device=pl]system.bit
  [destination_cpu=r5-1]app.elf
]
```

**Note:** The option split mode normal is same as the command line option split. This command line option is schedule to be deprecated.

---

**spkfile**

**Syntax**

```
[spkfile] <key filename>
```

**Description**

The Secondary Public Key (SPK) is used to authenticate partitions in the boot image. For more information, see Using Authentication.

**Arguments**

Specified file name.

**Example**

```
all:
[
  [pskfile] primarykey.pem
  [spkfile] secondarykey.pem
  [sskfile] secondarykey.pem
  [bootloader, authentication=rsa] fsbl.elf
  [authentication=rsa] hello.elf
]
```

**Note:** The secret key file contains the public key component of the key. You need not specify public key (SPK) when the secret key (SSK) is mentioned.
spksignature

**Code Example**

```bash
[spksignature] <Signature file>
```

**Description**

Imports SPK signature into the Authentication Certificate. This can be when the user does not want to share the secret key PSK, the user can create a signature and provide it to Bootgen.

**Arguments**

Specified file name.

**Example**

```bash
all:
[
    [ppkfile] ppk.txt
    [spkfile] spk.txt
    [headersignature] headers.sha256.sig
    [spksignature] spk.txt.sha256.sig
    [bootloader, authentication=rsa] fsbl.elf
]
```

spk_select

**Syntax**

```bash
[spk_select = <options>]
or
[auth_params] spk_select = <options>
```

**Description**

Options are:

- `spk-efuse`: Indicates that spk_id eFUSE is used for that partition. This is the default value.
- `user-efuse`: Indicates that user eFUSE is used for that partition.

Partitions loaded by CSU ROM will always use spk_efuse.

*Note*: The `spk_id` eFUSE specifies which key is valid. Hence, the ROM checks the entire field of `spk_id` eFUSE against the SPK ID to make sure its a bit for bit match.
The user eFUSE specifies which key ID is not valid (has been revoked). Hence, the firmware (non-ROM) checks to see if a given user eFUSE that represents the SPK ID has been programmed. `spk_select = user-efuse` indicates that user eFUSE will be used for that partition.

Example

```plaintext
the_ROM_image:
[
  [auth_params]ppk_select = 0
  [pskfile]psk.pem
  [sskfile]ssk1.pem

  [bootloader,
   authentication = rsa,
   spk_select = spk-efuse,
   spk_id = 0x12345678,
   sskfile = ssk2.pem
  ] zynqmp_fsbl.elf

  [destination_cpu =a53-0,
   authentication = rsa,
   spk_select = user-efuse,
   spk_id = 200,
   sskfile = ssk3.pem
  ] application1.elf

  [destination_cpu =a53-0,
   authentication = rsa,
   spk_select = spk-efuse,
   spk_id = 0x12345678,
   sskfile = ssk4.pem
  ] application2.elf
]
```

**sskfile**

**Syntax**

```plaintext
[sskfile] <key filename>
```

**Description**

The SSK - Secondary Secret Key key is used to authenticate partitions in the boot image. For more information, see Using Authentication

**Arguments**

Specified file name.
Example

all:
{
    [pskfile] primarykey.pem
    [sskfile] secondarykey.pem
    [bootloader, authentication=rsa] fsbl.elf
    [authentication=rsa] hello.elf
}

Note: The secret key file contains the public key component of the key. You need not specify the public key (PPK) when the secret key (PSK) is mentioned.

startup

Syntax

[startup=<address_value>] <partition>

Description

This option sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute.

Example

all:
{
    [bootloader] fsbl.elf
    [startup=0x1000000] app.elf
}

trustzone

Syntax

[trustzone=<options>] <partition>

Description

Configures the core to be TrustZone secure or nonsecure. Options are:

- secure
- nonsecure (default)
Example

```plaintext
e: all:
    [bootloader,destination_cpu=a53-0] fsbl.elf
    [exception_level=el-3,trustzone = secure] bl31.elf
```

## udf_bh

### Syntax

```
[udf_bh] <filename>
```

### Description

Imports a file of data to be copied to the user defined field (UDF) of the Boot Header. The input user defined data is provided through a text file in the form of a hex string. Total number of bytes in UDF in Xilinx® SoCs:

- zynq: 76 bytes
- zynqmp: 40 bytes

### Arguments

Specified file name.

### Example

```plaintext
e: all:
    [udf_bh] test.txt
    [bootloader] fsbl.elf
    hello.elf
```

The following is an example of the input file for udf_bh:

**Sample input file for udf_bh - test.txt**

```
123456789abcdef85072696e63653030301440408706d616c6164000508
266431530102030405060708090a0b0c0d0e0f101112131415161718191a1b
1c1d1
```
**udf_data**

**Syntax**

```
[udf_data=<filename>] <partition>
```  

**Description**

Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate. For more information, see Authentication for more information about authentication certificates.

**Arguments**

Specified file name.

**Example**

```
all:
{
    [pskfile] primary0.pem
    [sskfile] secondary0.pem
    [bootloader, destination_cpu=a53-0,
      authentication=rsa, udf_data=udf.txt] fsbl.elf
    [destination_cpu=a53-0, authentication=rsa] hello.elf
}
```  

**xip_mode**

**Syntax**

```
[xip_mode] <partition>
```  

**Description**

Indicates 'eXecute In Place' for FSBL to be executed directly from QSPI flash.

**Note:** This attribute is only applicable for an FSBL/Bootloader partition.

**Arguments**

Specified partition.
Example

This example shows how to create a boot image that executes in place for a Zynq® UltraScale+™ MPSoC device.

```
all:
[
  [bootloader, xip_mode] fsbl.elf
  application.elf
]
```
Appendix C

Command Reference

arch

Syntax

-arch [options]

Description

Xilinx® family architecture for which the boot image needs to be created.

Arguments

• zynq: Zynq®-7000 device architecture. This is the default value.
• zynqmp: Zynq® UltraScale+™ MPSoC device architecture.
• fpga: Image is targeted for other FPGA architectures.

Return Value

None

Example

bootgen -arch zynq -image test.bif -o boot.bin

bif_help

Syntax

bootgen -bif_help

bootgen -bif_help aeskeyfile
Description

Lists the supported BIF file attributes. For a more detailed explanation of each bif attribute, specify the attribute name as argument to \texttt{-bif\_help} on the command line.

---

dual\_qspi\_mode

Syntax

\texttt{bootgen -dual\_qspi\_mode [parallel] | [stacked <size>]}  

Description

Generates two output files for dual QSPI configurations. In the case of stacked configuration, size (in MB) of the flash needs to be mentioned (16 or 32 or 64 or 128).

Examples

This example generates two output files for independently programming to both flashes in QSPI dual parallel configuration.

\texttt{bootgen -image test.bif -o -boot.bin -dual\_qspi\_mode parallel}

This example generates two output files for independently programming to both flashes in a QSPI dual stacked configuration. The first 64 MB of the actual image is written to first file and the remainder to the second file. In case the actual image itself is less than 64 MB, only one file is generated.

\texttt{bootgen -image test.bif -o -boot.bin -dual\_qspi\_mode stacked 64}

Arguments

- parallel
- stacked <size>

---

efuseppkbits

Syntax

\texttt{bootgen -image test.bif -o boot.bin -efuseppkbits efusefile.txt}
Arguments

efusefile.txt

Description

This option specifies the name of the eFUSE file to be written to contain the PPK hash. This option generates a direct hash without any padding. The efusefile.txt file is generated containing the hash of the PPK key. Where:

- Zynq®-7000 uses the SHA2 protocol for hashing.
- Zynq® UltraScale+™ MPSoC uses the SHA3 for hashing.

encrypt

Syntax

```
bootgen -image test.bif -o boot.bin -encrypt <efuse|bbram|>  
```

Description

This option specifies how to perform encryption and where the keys are stored. The NKY key file is passed through the BIF file attribute aeskeyfile. Only the source is specified using command line.

Arguments

Key source arguments:

- efuse: The AES key is stored in eFUSE. This is the default value.
- bbram: The AES key is stored in BBRAM.

encryption_dump

Syntax

```
bootgen -arch zynqmp -image test.bif -encryption_dump  
```
Description

Generates an encryption log file, `aes_log.txt`. The `aes_log.txt` generated has the details of AES Key/IV pairs used for encrypting each block of data. It also logs the partition and the AES key file used to encrypt it.

Note: This option is supported only for Zynq® UltraScale+™ MPSoC

Example

```yaml
all:
  - [bootloader, encryption=aes, aeskeyfile=test.nky] fsbl.elf
  - [encryption=aes, aeskeyfile=test1.nky] hello.elf
```

fill

Syntax

```
bootgen -arch zynq -image test.bif -fill 0xAB -o boot.bin
```

Description

This option specifies the byte to use for filling padded/reserved memory in `<hex byte>` format.

Outputs

The `boot.bin` file in the 0xAB byte.

Example

The output image is generated with name `boot.bin`. The format of the output image is determined based on the file extension of the file given with `-o` option, where `-fill`: Specifies the Byte to be padded. The `<hex byte>` is padded in the header tables instead of 0xFF.

```
bootgen -arch zynq -image test.bif -fill 0xAB -o boot.bin
```

generate_hashes

Syntax

```
bootgen -image test.bif -generate_hashes
```
Description

This option generates hash files for all the partitions and other components to be signed like boot header, image and partition headers. This option generates a file containing PKCS#1v1.5 padded hash for the Zynq®-7000 format:

Table 23: Zynq: SHA-2 (256-bytes)

<table>
<thead>
<tr>
<th>Value</th>
<th>SHA-2 Hash*</th>
<th>T-Padding</th>
<th>0x0</th>
<th>0xFF</th>
<th>0x01</th>
<th>0x00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bytes</td>
<td>32</td>
<td>19</td>
<td>1</td>
<td>202</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This option generates the file containing PKCS#1v1.5 padded hash for the Zynq® UltraScale+™ MPSoC format:

Table 24: ZynqMP: SHA-3 (384-bytes)

<table>
<thead>
<tr>
<th>Value</th>
<th>0x0</th>
<th>0x1</th>
<th>0xFF</th>
<th>0xFF</th>
<th>T-Padding</th>
<th>SHA-3 Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bytes</td>
<td>1</td>
<td>1</td>
<td>314</td>
<td>1</td>
<td>19</td>
<td>48</td>
</tr>
</tbody>
</table>

Example

test:
[
  [pskfile] ppk.txt
  [sskfile] spk.txt
  [bootloader, authentication=rsa] fsbl.elf
  [authentication=rsa] hello.elf
]

Bootgen generates the following hash files with the specified BIF:

- bootheader hash
- spk hash
- header table hash
- fsbl.elf partition hash
- hello.elf partition hash

generate_keys

Syntax

```
bootgen -image test.bif -generate_keys <rsa|pem|obfuscated>
```
Description

This option generates keys for authentication and obfuscated key used for encryption.

*Note:* For more information on generating encryption keys, see [Key Generation](#).

**Authentication Key Generation Example**

Authentication key generation example. This example generates the authentication keys in the paths specified in the BIF file.

**Examples**

```plaintext
image:
 [
  [ppkfile] <path/ppkgenfile.txt>
  [pskfile] <path/pskgenfile.txt>
  [spkfile] <path/spkgenfile.txt>
  [sskfile] <path/sskgenfile.txt>
]
```

**Obfuscated Key Generation Example**

This example generates the obfuscated in the same path as that of the `familykey.txt`.

**Command:**

```plaintext
bootgen -image test.bif -generata_keys rsa
```

The Sample BIF file is shown in the following example:

```plaintext
image:
 [
  [aeskeyfile] aes.nky
  [bh_key_iv] bhkeyiv.txt
  [familykey] familykey.txt
]
```

**Arguments**

- rsa
- pem
- obfuscated
image

Syntax

-image <BIF_filename>

Description

This option specifies the input BIF file name. The BIF file specifies each component of the boot image in the order of boot and allows optional attributes to be specified to each image component. Each image component is usually mapped to a partition, but in some cases an image component can be mapped to more than one partition if the image component is not contiguous in memory.

Arguments

bif_filename

Example

bootgen -arch zynq -image test.bif -o boot.bin

The Sample BIF file is shown in the following example:

```plaintext
the_ROM_image:
{
    [init] init_data.int
    [bootloader] fsbl.elf
    Partition1.bit
    Partition2.elf
}
```

log

Syntax

bootgen -image test.bif -o -boot.bin -log trace

Description

Generates a log while generating the boot image. There are various options for choosing the level of information. The information is displayed on the console as well as in the log file, named bootgen_log.txt is generated in the current working directory.
Arguments

- error: Only the error information is captured.
- warning: The warnings and error information is captured. This is the default value.
- info: The general information and all the above info is captured.
- trace: More detailed information is captured along with the information above.

nonbooting

Syntax

```bash
bootgen -arch zynq -image test.bif -o test.bin -nonbooting
```

Description

This option is used to create an intermediate boot image. An intermediate `test.bin` image is generated as output even in the absence of secret key, which is required to generate an authenticated image. This intermediate image cannot be booted.

Example

```bash
all:
  [
    [ppkfile]primary.pub
    [spkfile]secondary.pub
    [spksignature]secondary.pub.sha256.sig
    [bootimage,authentication=rsa,presign=fsbl_0.elf.0.sha256.sig]fsbl_e.bin
  ]
```

O

Syntax

```bash
bootgen -arch zynq -image test.bif -o boot.<bin|mcs>
```

Description

This option specifies the name of the output image file with a `.bin` or `.mcs` extension.
Outputs

A full boot image file in either BIN or MCS format.

Example

```
bootgen -arch zynq -image test.bif -o boot.mcs
```

The boot image is output in an MCS format.

Syntax

```
bootgen -image test.bif -o boot.bin -p xc7z020clg48 -encrypt efuse
```

Description

This option specifies the partname of the Xilinx® device. This is needed for generating an encryption key. It is copied verbatim to the *.nky file in the Device line of the nky file. This is applicable only when encryption is enabled. If the key file is not present in the path specified in BIF file, then a new encryption key is generated in the same path and xc7z020clg48 is copied along side the Device field in the nky file. The generated image is an encrypted image.

padimageheader

Syntax

```
bootgen -image test.bif -w on -o boot.bin -padimageheader=<0|1>
```

Description

This option pads the Image Header Table and Partition Header Table to maximum partitions allowed, to force alignment of following partitions. This feature is enabled by default. Specifying a 0 disables this feature. The boot.bin has the image header tables and partition header tables in actual and no extra tables are padded. If nothing is specified or if -padimageheader=1, the total image header tables and partition header tables are padded to max partitions.

Arguments

- 1: Pad the header tables to max partitions. This is the default value.
• 0: Do not pad the header tables.

**Image or Partition Header Lengths**

- zynq: Maximum Partitions - 14
- zynqmp: Maximum Partitions - 32

---

**process_bitstream**

**Syntax**

```
-process_bitstream <bin|mcs>
```

**Description**

Processes only the bitstream from the BIF and outputs it as an MCS or a BIN file. For example: If encryption is selected for bitstream in the BIF file, the output is an encrypted bitstream.

**Arguments**

- `bin`: Output in BIN format.
- `mcs`: Output in MCS format.

**Returns**

Output generated is bitstream in BIN or MCS format; a processed file without any headers attached.

---

**read**

**Syntax**

```
-read [options]
```

**Description**

Used to read boot headers, image headers, and partition headers based on the options.

**Arguments**

- `bh`: To read boot header from bootimage in human readable form
• iht: To read image header table from bootimage
• ih: To read image headers from bootimage.
• pht: To read partition headers from bootimage
• bootgen -arch zynqmp -read BOOT.bin
• ac: To read authentication certificates from bootimage

spksignature

Syntax

```bash
bootgen -image test.bif -w on -o boot.bin -spksignature spksignfile.txt
```

Description

This option is used to generate the SPK signature file. This option must be used only when `spkfile` and `pskfile` are specified in BIF. The SPK signature file (spksignfile.txt) is generated.

Option

Specifies the name of the signature file to be generated.

split

Syntax

```bash
bootgen -arch zynq -image test.bif -split bin
```

Description

This option outputs each data partition with headers as a new file in MCS or BIN format.

Outputs

Output files generated are:

- Bootheader + Image Headers + Partition Headers + Fsbl.elf
- Partition1.bit
- Partition2.elf
Example

```plaintext
the_ROM_image:
{
    [bootloader] Fsbl.elf
    Partition1.bit
    Partition2.elf
}
```

verify

**Syntax**

```
bootgen -arch zynqmp -verify boot.bin
```

**Description**

This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image will be verified against the available partitions. Verification is performed in the following steps:

1. Verify Header Authentication Certificate, verify SPK Signature, and verify Header Signature.
2. Verify Bootloader Authentication Certificate, verify Boot Header Signature, verify SPK Signature, and verify Bootloader Signature

This is repeated for all partitions in the given boot image.

verify_kdf

**Syntax**

```
bootgen -arch zynqmp -verify_kdf testVec.txt
```
Description
The format of the testVec.txt file is as below.

L = 256
KI = d54b6fd94f7cf98fd955517f937e9927f9536caeb148fba1818c1ba46bba3a4
FixedInputDataByteLen = 60
FixedInputData =
94c4a0c69526196c1377cebf0a2ae0fb4b57797c61bea8eeb0518ca08652d14a5e1bd1b116b1
794ac8a476acbdacb0d4f6142d7b8515bad09ec72f7af

Bootgen uses the counter Mode KDF to generate the output key (KO) based on the given input data in the test vector file. This KO will be printed on the console for the user to compare.

W
Syntax
bootgen -image test.bif -w on -o boot.bin
or
bootgen -image test.bif -w -o boot.bin

Description
This option specifies whether to overwrite an existing file or not. If the file boot.bin already exists in the path, then it is overwritten. Options -w on and -w are treated as same. If the -w option is not specified, the file will not be overwritten by default.

Arguments
• on: Specified with the -w on command with or -w with no argument. This is the default value.
• off: Specifies to not overwrite an existing file.

zynqmpes1
Syntax
bootgen -arch zynqmp -image test.bif -o boot.bin -zynqmpes1
**Description**

This option specifies that the image generated will be used on ES1 (1.0). This option makes a difference only when generating an Authenticated image; otherwise, it is ignored. The default padding scheme is for (2.0) ES2 and above.

---

**Initialization Pairs and INT File Attribute**

Initialization pairs let you easily initialize Processor Systems (PS) registers for the MIO multiplexer and flash clocks. This allows the MIO multiplexer to be fully configured before the FSBL image is copied into OCM or executed from flash with eXecute in place (XIP), and allows for flash device clocks to be set to maximum bandwidth speeds.

There are 256 initialization pairs at the end of the fixed portion of the boot image header. Initialization pairs are designated as such because a pair consists of a 32-bit address value and a 32-bit data value. When no initialization is to take place, all of the address values contain 0xFFFFFFFF, and the data values contain 0x00000000. Set initialization pairs with a text file that has an .int file extension by default, but can have any file extension.

The [init] file attribute precedes the file name to identify it as the INIT file in the BIF file. The data format consists of an operation directive followed by:

- An address value
- an = character
- a data value

The line is terminated with a semicolon (;). This is one .set operation directive; for example:

```
.set. 0xE0000018 = 0x00000411; // This is the 9600 uart setting.
```

Bootgen fills the boot header initialization from the INT file up to the 256 pair limit. When the BootROM runs, it looks at the address value. If it is not 0xFFFFFFFF, the BootROM uses the next 32-bit value following the address value to write the value of address. The BootROM loops through the initialization pairs, setting values, until it encounters a 0xFFFFFFFF address, or it reaches the 256th initialization pair.

Bootgen provides a full expression evaluator (including nested parenthesis to enforce precedence) with the following operators:

- *= multiply/divide
- %= mod
- an address value
- u/ divide

---

**Appendix C: Command Reference**

**UG1283 (v2019.2) October 30, 2019  Bootgen User Guide**

[www.xilinx.com](http://www.xilinx.com)
The numbers can be hex (0x), octal (0o), or decimal digits. Number expressions are maintained as 128-bit fixed-point integers. You can add white space around any of the expression operators for readability.
CAUTION! This utility has been deprecated. Instead, use the -read option.

The bootgen_utility is a tool used to dump the contents of a Boot Image generated by Bootgen, into a human-readable log file. This is useful in debugging and understanding the contents of the different header tables of a boot image.

The utility generates the following files as output:

- Dump of all header tables.
- Dump of register init table.
- Dump of individual partitions.

Note: If the partitions are encrypted, the dump will be the encrypted partition and not the decrypted one.

Usage:

```
bootgen_utility
   -arch <zynq | zynqmp> -bin <binary input file name> -out <output text file>
```

Example:

```
bootgen_utility
   -arch zynqmp -bin boot.bin -out info.txt
```

Sample output file looks like the following:
### Example Output

<table>
<thead>
<tr>
<th>Flash Address</th>
<th>Offset</th>
<th>Description</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0x00000000]</td>
<td>[0x00]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x00000004]</td>
<td>[0x04]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x00000008]</td>
<td>[0x08]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x0000000c]</td>
<td>[0x0c]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x00000010]</td>
<td>[0x10]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x00000014]</td>
<td>[0x14]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x00000018]</td>
<td>[0x18]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x0000001c]</td>
<td>[0x1c]</td>
<td>ARM Vector Table</td>
<td>0xeafffffe</td>
</tr>
<tr>
<td>[0x00000020]</td>
<td>[0x20]</td>
<td>Width Detection Word</td>
<td>0xaaaa5556</td>
</tr>
<tr>
<td>[0x00000024]</td>
<td>[0x24]</td>
<td>Header Signature</td>
<td>0x88c6e68</td>
</tr>
<tr>
<td>[0x00000028]</td>
<td>[0x28]</td>
<td>Encryption Key Source</td>
<td>0x00 (Not Encrypted)</td>
</tr>
<tr>
<td>[0x0000002c]</td>
<td>[0x2c]</td>
<td>Header Version</td>
<td>0x1010000</td>
</tr>
<tr>
<td>[0x00000030]</td>
<td>[0x30]</td>
<td>Load Image Byte Offset</td>
<td>0x2000000</td>
</tr>
<tr>
<td>[0x00000034]</td>
<td>[0x34]</td>
<td>Load Image Byte Length</td>
<td>0x0000000</td>
</tr>
<tr>
<td>[0x00000038]</td>
<td>[0x38]</td>
<td>Image Load Byte Address</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x0000003c]</td>
<td>[0x3c]</td>
<td>Image Execution Byte Address</td>
<td>0x0f200000</td>
</tr>
<tr>
<td>[0x00000040]</td>
<td>[0x40]</td>
<td>Total Image Byte Length</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x00000044]</td>
<td>[0x44]</td>
<td>QSPI Config Word</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x00000048]</td>
<td>[0x48]</td>
<td>Header Checksum</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x00000050]</td>
<td>[0x50]</td>
<td>Image Header Table Offset</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x00000054]</td>
<td>[0x54]</td>
<td>Partition Header Table Offset</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

---

**INITIALISATION TABLE**

Refer file "info_boot_new_register_init_table.txt"

---

**IMAGE HEADER TABLE**

<table>
<thead>
<tr>
<th>Flash Address</th>
<th>Offset</th>
<th>Description</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0x00000000]</td>
<td>[0x00]</td>
<td>Version</td>
<td>0x10200000</td>
</tr>
<tr>
<td>[0x00000004]</td>
<td>[0x04]</td>
<td>Count of Image Headers</td>
<td>0x01000000</td>
</tr>
<tr>
<td>[0x00000008]</td>
<td>[0x08]</td>
<td>Offset to 1st Partition Header</td>
<td>0x20000000</td>
</tr>
<tr>
<td>[0x0000000c]</td>
<td>[0x0c]</td>
<td>Offset to 1st Image Header</td>
<td>0x24000000</td>
</tr>
<tr>
<td>[0x00000010]</td>
<td>[0x10]</td>
<td>Offset to Header Auth. Cert.</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

---

**IMAGE HEADERS**

<table>
<thead>
<tr>
<th>Flash Address</th>
<th>Offset</th>
<th>Description</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0x00000000]</td>
<td>[0x00]</td>
<td>Next Image Header Pointer</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x00000004]</td>
<td>[0x04]</td>
<td>Next 1st Partition Header Pointer</td>
<td>0x00200000</td>
</tr>
<tr>
<td>[0x00000008]</td>
<td>[0x08]</td>
<td>Partition Count (Wrong Info)</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[0x0000000c]</td>
<td>[0x0c]</td>
<td>Image Length (Wrong Info)</td>
<td>0x01000000</td>
</tr>
<tr>
<td>[0x00000010]</td>
<td>[0x10]</td>
<td>Image Name</td>
<td>fsl_xip_elf</td>
</tr>
</tbody>
</table>
Additional Resources and Legal Notices

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select Help → Documentation and Tutorials.
- On Windows, select Start → All Programs → Xilinx Design Tools → DocNav.
- At the Linux command prompt, enter docnav.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the Design Hubs View tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.
Additional Resources

Bootgen Document Resources

The following documents provide additional information on the processes that support the Bootgen process:

- For Zynq®-7000 SoC:
  - *Zynq-7000 SoC Software Developers Guide* (UG821)

- For Zynq® UltraScale+™ MPSoC:
  - *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)

Other Document Resources for Bootgen

- *Zynq-7000 SoC Secure Boot Getting Started Guide* (UG1025)
- *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* (UG1209)
- *Sector Boot of Zynq-7000 SoC* (XAPP1175)
- *Run Time Integrity and Authentication Check of Zynq-7000 SoC System Memory* (XAPP1225)
- *Programming BBRAM and eFUSEs* (XAPP1319)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the...
Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2018-2019 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. All other trademarks are the property of their respective owners.