

# Bootgen User Guide

UG1283 (v2023.1) June 15, 2023

AMD Adaptive Computing is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this [link](#) for more information.



# Table of Contents

<b>Chapter 1: Introduction.....</b>	<b>7</b>
Installing Bootgen.....	8
Boot Time Security.....	8
<b>Chapter 2: Boot Image Layout.....</b>	<b>9</b>
Zynq 7000 SoC Boot and Configuration.....	9
Zynq UltraScale+ MPSoC Boot and Configuration.....	19
Versal Adaptive SoC Boot Image Format.....	32
<b>Chapter 3: Creating Boot Images.....</b>	<b>46</b>
Boot Image Format (BIF).....	46
BIF Syntax and Supported File Types.....	47
Attributes.....	52
<b>Chapter 4: Using Bootgen GUI.....</b>	<b>63</b>
Launch Bootgen GUI .....	63
Bootgen GUI for Zynq 7000 and Zynq UltraScale+ Devices.....	63
Using Bootgen GUI Options for Versal Adaptive SoCs.....	65
Using Bootgen on the Command Line.....	67
Commands and Descriptions.....	67
<b>Chapter 5: Boot Time Security.....</b>	<b>71</b>
Using Encryption.....	72
Using Authentication.....	84
Versal Authentication Support.....	95
Versal Hashing Scheme.....	96
Using HSM Mode.....	97
Creating a Keystone B Device Boot Image Using HSM .....	126
<b>Chapter 6: SSIT Support.....</b>	<b>127</b>
<b>Chapter 7: FPGA Support.....</b>	<b>139</b>

Encryption and Authentication.....	139
HSM Mode.....	140
HSM Flow with Both Authentication and Encryption.....	143
<b>Chapter 8: Use Cases and Examples.....</b>	<b>145</b>
Zynq MPSoC Use Cases.....	145
Versal Adaptive SoC Use Cases.....	155
<b>Chapter 9: BIF Attribute Reference.....</b>	<b>166</b>
aarch32_mode.....	166
aeskeyfile.....	167
alignment.....	170
auth_params.....	171
authentication.....	173
big_endian.....	175
bbram_kek_iv.....	176
bh_kek_iv.....	176
bh_keyfile.....	176
bh_key_iv.....	178
bhsignature.....	178
blocks.....	179
boot_config.....	181
boot_device.....	182
bootimage.....	184
bootloader.....	186
bootvectors.....	187
checksum.....	187
copy.....	188
core.....	189
delay_auth.....	190
delay_handoff.....	190
delay_load.....	191
destination_cpu.....	192
destination_device.....	193
early_handoff.....	193
efuse_kek_iv.....	194
efuse_user_kek0_iv.....	194
efuse_user_kek1_iv.....	195
encryption.....	195

exception_level.....	197
familykey.....	198
file.....	199
fsbl_config.....	199
headersignature.....	200
hivec.....	201
id.....	202
image.....	204
imagestore.....	204
init.....	205
keysrc.....	206
keysrc_encryption.....	207
load.....	208
metaheader.....	209
name.....	210
offset.....	211
overlay_cdo.....	212
parent_id.....	212
partition.....	213
partition_owner, owner.....	214
pid.....	215
pmufw_image.....	215
ppkfile.....	216
presign.....	217
pskfile.....	218
puf_file.....	219
reserve.....	220
split.....	221
spkfile.....	222
spksignature.....	223
spk_select.....	224
sskfile.....	226
startup.....	227
trustzone.....	228
type.....	229
udf_bh.....	229
udf_data.....	230
userkeys.....	231

xip_mode.....	233
<b>Chapter 10: Command Reference.....</b>	<b>234</b>
arch.....	234
authenticatedjtag.....	235
bif_help.....	235
dual_ospi_mode.....	235
dual_qspi_mode.....	236
dump.....	237
dump_dir.....	237
efuseppkbits.....	238
encrypt.....	238
encryption_dump.....	239
fill.....	239
generate_hashes.....	240
generate_keys.....	241
h, help.....	242
image.....	242
log.....	243
nonbooting.....	243
o.....	244
p.....	244
padimageheader.....	245
process_bitstream.....	245
read.....	246
spksignature.....	247
split.....	247
verify.....	248
verify_kdf.....	248
w.....	249
zynqmpes1.....	249
Initialization Pairs and INT File Attribute.....	249
<b>Chapter 11: CDO Utility.....</b>	<b>251</b>
Accessing.....	251
Usage.....	251
Examples.....	252
<b>Chapter 12: Design Advisories for Bootgen.....</b>	<b>254</b>



**Appendix A: Additional Resources and Legal Notices..... 255**

Finding Additional Documentation..... 255

Support Resources..... 256

Additional Resources..... 256

Revision History..... 257

Please Read: Important Legal Notices..... 257

# Introduction

AMD FPGAs, system-on-chip (SoC) devices, and adaptive SoCs 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 an AMD tool that lets you stitch binary files together and generate device boot images. Bootgen defines multiple properties, attributes and parameters that are put in while creating boot images for use in an AMD device.

The secure boot feature for AMD 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. Bootgen 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 AMD Vitis™ Integrated Development Environment (IDE) for generating basic boot images, 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 AMD SoC and adaptive SoC, Bootgen has the ability to encrypt and authenticate partitions for AMD 7 series and later FPGAs, as described in [Chapter 7: 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 in the Vitis IDE for simple boot image creation, or in command-line mode for more complex boot images. You can install Bootgen from the AMD Unified Installer for FPGAs and Adaptive SoCs.

After you install Vitis with Bootgen, you can start and use the tool from the Vitis IDE option that contains the most common actions for rapid development and experimentation, or from the XSCT (Software Command Tool).

The command line option provides many more options for creating a boot image. See [Using Bootgen Options on the Command Line](#) for more information.

---

## Boot Time Security

Secure booting through latest authentication methods is supported to prevent unauthorized or modified code from being run on AMD 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](#)).
- *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).
- *Versal Adaptive SoC Technical Reference Manual* ([AM011](#)).

**Note:** For additional information, see the *Versal Adaptive SoC Security Manual* (UG1508). This manual requires an active NDA to be downloaded from the [Design Security Lounge](#).

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.



# 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 AMD Zynq™ UltraScale+™ MPSoC devices, see [Zynq UltraScale+ MPSoC Boot and Configuration](#).
- For information on how to use Bootgen for AMD FPGAs, see [Chapter 7: FPGA Support](#).
- For information on AMD Versal™ adaptive SoC, see [Versal Adaptive SoC Boot Image Format](#).

Building a boot image involves the following steps:

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

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

The input files are not necessarily different for each device (for example, for every device, ELF files can be input files that can be part of the boot image), but the format of the boot image is different. The following sections 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 AMD Zynq™ 7000 SoC devices. 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-7000 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 an AMD Zynq™ 7000 SoC boot image.

Figure 1: Boot Header

Boot Header			
Register Initialization Table			
Image Header Table			
Image Header 1	Image Header 2	---	Image Header n
Partition Header 1	Partition Header 2	---	Partition Header n
Header Authentication Certificate (Optional)			
Partition 1 (FSBL)			AC (Optional)
Partition 2			AC (Optional)
.			
Partition n			AC (Optional)

X25912-102521

## Zynq 7000 SoC Boot Header

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.

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 determine where the 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 Image Header Table](#)
- [Zynq 7000 SoC Image Header](#)
- [Zynq 7000 SoC Partition Header](#)
- [Zynq 7000 SoC Authentication Certificate](#)

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

**Table 1: Zynq 7000 SoC Boot Header**

Address Offset	Parameter	Description
0x00-0x1F	Arm® Vector table	Filled with dummy vector table by Bootgen (Arm Op code 0xEAFFFFFFFFE, which is a branch-to-self infinite loop intended to catch uninitialized vectors.
0x20	Width Detection Word	This is required to identify the QSPI flash in single/dual stacked or dual parallel mode. 0xAA995566 in little endian format.
0x24	Header Signature	Contains 4 bytes 'X','N','L','X' in byte order, which is 0x584c4e58 in little endian format.
0x28	Key Source	Location of encryption key within the device:  0x3A5C3C5A: Encryption key in BBRAM. 0xA5C3C5A3: Encryption key in eFUSE. 0x00000000: Not Encrypted.
0x2C	Header Version	0x01010000
0x30	Source Offset	Location of FSBL (bootloader) in this image file.
0x34	FSBL Image Length	Length of the FSBL, after decryption.
0x38	FSBL Load Address (RAM)	Destination RAM address to which to copy the FSBL.
0x3C	FSBL Execution address (RAM)	Entry vector for FSBL execution.
0x40	Total FSBL Length	Total size of FSBL after encryption, including authentication certificate (if any) and padding.
0x44	QSPI Configuration Word	Hard coded to 0x00000001.
0x48	Boot Header Checksum	Sum of words from offset 0x20 to 0x44 inclusive. The words are assumed to be little endian.
0x4c-0x97	User Defined Fields	76 bytes
0x98	Image Header Table Offset	Pointer to Image Header Table
0x9C	Partition Header Table Offset	Pointer to Partition Header 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

Address Offset	Parameter	Description
0xA0 to 0x89C	Register Initialization Pairs: <address>:<value>:	Address = 0xFFFFFFFF means skip that register and ignore the value. All the unused register fields must be set to Address=0xFFFFFFFF and value = 0x0.

## 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 that 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 AMD Zynq™ 7000 SoC device.

Table 3: Zynq 7000 SoC Image Header Table

Address Offset	Parameter	Description
0x00	Version	0x01010000: Only fields available are 0x0, 0x4, 0x8, 0xC, and a padding 0x01020000:0x10 field is added.
0x04	Count of Image Headers	Indicates the number of image headers.
0x08	First Partition Header Offset	Pointer to first partition header. (word offset)
0x0C	First Image Header Offset	Pointer to first image header. (word offset)
0x10	Header Authentication Certificate Offset	Pointer to the authentication certificate header. (word offset)
0x14	Reserved	Defaults to 0xFFFFFFFF.

## 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 also contains the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the AMD Zynq™ 7000 SoC device.

Table 4: Zynq-7000 SoC Image Header

Address Offset	Parameter	Description
0x00	Next Image Header	Link to next Image Header. 0 if last Image Header (word offset).
0x04	Corresponding partition header	Link to first associated Partition Header (word offset).
0x08	Reserved	Always 0.
0x0C	Partition Count Length	Number of partitions associated with this image.
0x10 to N	Image Name	Packed in big endian order. To reconstruct the string, unpack 4 bytes at a time, reverse the order, and concatenate. For example, the string "FSBL10.ELF" is packed as 0x10: 'L', 'B', 'S', 'F', 0x14: 'E', '.', '0', '1', 0x18: '\0', '\0', 'F', 'L'. The packed image name is a multiple of 4 bytes.
N	String Terminator	0x00000000
N+4	Reserved	Defaults to 0xFFFFFFFF to 64 bytes boundary.

## 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 AMD Zynq™ 7000 SoC Partition Header.

**Note:** An ELF file with three loadable sections has one image header and three partition header tables.

Table 5: Zynq 7000 SoC Partition Header

Offset	Name	Notes
0x00	Encrypted Partition length	Encrypted partition data length.
0x04	Unencrypted Partition length	Unencrypted data length.
0x08	Total partition word length (Includes Authentication Certificate.) See <a href="#">Zynq 7000 SoC Authentication Certificate</a> .	The total partition word length comprises the encrypted information length with padding, the expansion length, and the authentication length.
0x0C	Destination load address.	The RAM address into which this partition is to be loaded.
0x10	Destination execution address.	Entry point of this partition when executed.
0x14	Data word offset in Image	Position of the partition data relative to the start of the boot image.
0x18	Attribute Bits	See <a href="#">Zynq 7000 SoC Partition Attribute Bits</a>

Table 5: Zynq 7000 SoC Partition Header (cont'd)

Offset	Name	Notes
0x1C	Section Count	Number of sections in a single partition.
0x20	Checksum Word Offset	Location of the corresponding checksum word in the boot image.
0x24	Image Header Word Offset	Location of the corresponding Image Header in the boot image.
0x28	Authentication Certification Word Offset	Location of the corresponding Authentication Certification in the boot image.
0x2C-0x38	Reserved	Reserved
0x3C	Header Checksum	Sum of the previous words in the Partition Header.

### Zynq 7000 SoC Partition Attribute Bits

The following table describes the Partition Attribute bits of the partition header table for an AMD Zynq™ 7000 SoC device.

Table 6: Zynq 7000 SoC Partition Attribute Bits

Bit Field	Description	Notes
31:18	Reserved	Not used
17:16	Partition owner	0: FSBL 1: UBOOT 2 and 3: reserved
15	RSA signature present	0: No RSA authentication certificate 1: RSA authentication certificate
14:12	Checksum type	0: None 1: MD5 2-7: reserved
11:8	Reserved	Not used
7:4	Destination device	0: None 1: PS 2: PL 3: INT 4-15: Reserved
3:2	Reserved	Not used
1:0	Reserved	Not used

## 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 needs authentication. Header Table Authentication Certificate is appended at end of the header tables content.

The AMD 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 an AMD Zynq™ 7000 SoC is as shown in the following table.

**Table 7: Zynq 7000 SoC Authentication Certificate**

Authentication Certificate Bits		Description
0x00	Authentication Header = 0x0101000. See <a href="#">Zynq 7000 SoC Authentication Certificate Header</a> .	
0x04	Certificate size	
0x08	UDF (56 bytes)	
0x40	PPK	Mod (256 bytes)
0x140		Mod Ext (256 bytes)
0x240		Exponent
0x244		Pad (60 bytes)
0x280	SPK	Mod (256 bytes)
0x380		Mod Ext (256 bytes)
0x480		Exponent (4 bytes)
0x484		Pad (60 bytes)
0x4C0	SPK Signature = RSA-2048 (PSK, Padding    SHA-256 (SPK))	
0x5C0	FSBL Partition Signature = RSA-2048 (SSK, SHA256 (Boot Header    FSBL partition))	
0x5C0	Other Partition Signature = RSA-2048 (SSK, SHA-256 (Partition    Padding    Authentication Header    PPK    SPK    SPK Signature))	

### Zynq 7000 SoC Authentication Certificate Header

The following table describes the AMD Zynq™ 7000 SoC Authentication Certificate Header.



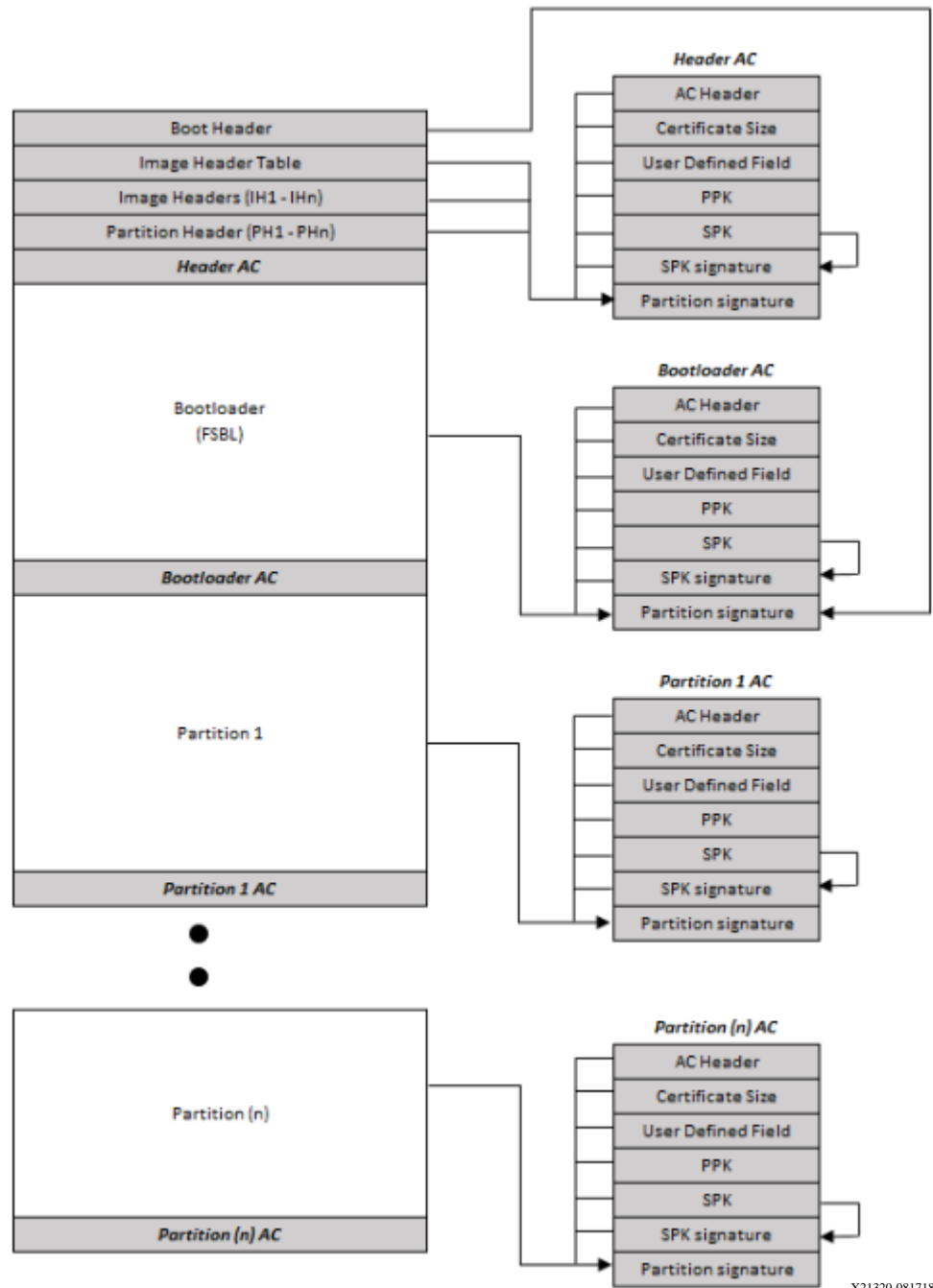
Table 8: Zynq 7000 SoC Authentication Certificate Header

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

## Zynq 7000 SoC Boot Image Block Diagram

The following is a diagram of the components that can be included in an AMD Zynq™ 7000 SoC boot image.

Figure 2: Zynq 7000 SoC Boot Image Block Diagram



X21320-081718

# Zynq UltraScale+ MPSoC Boot and Configuration

## Introduction

AMD 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 "Configuration Security Unit" in the *Zynq UltraScale+ MPSoC: Software Developers Guide* ([UG1137](#)).

## Zynq UltraScale+ MPSoC Boot Image

The following figure shows the AMD Zynq™ UltraScale+™ MPSoC boot image.

Figure 3: Zynq UltraScale+ MPSoC Boot Image

Boot Header			
Register Initialization Table			
PUF Helper Data (Optional)			
Image Header Table			
Image Header 1	Image Header 2	---	Image Header n
Partition Header 1	Partition Header 2	---	Partition Header n
Header Authentication Certificate (Optional)			
Partition 1 (FSBL)		PMU FW (Optional)	AC (Optional)
Partition 2			AC (Optional)
.			
Partition n			AC (Optional)

X23449-102919

## 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 AMD Zynq™ UltraScale+™ MPSoC device.

**Table 9: Zynq UltraScale+ MPSoC Device Boot Header**

Address Offset	Parameter	Description
0x00-0x1F	Arm® vector table	XIP ELF vector table:  0xEAFFFFF: for Cortex®-R5F and Cortex A53 (32-bit) 0x14000000: for Cortex A53 (64-bit)
0x20	Width Detection Word	This field is used for QSPI width detection. 0xAA995566 in little endian format.
0x24	Header Signature	Contains 4 bytes 'X', 'N', 'L', 'X' in byte order, which is 0x584c4e58 in little endian format.
0x28	Key Source	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)
0x2C	FSBL Execution address (RAM)	FSBL execution address in OCM or XIP base address.
0x30	Source Offset	If no PMUFW, then it is the start offset of FSBL. If PMUFW, then start of PMUFW.
0x34	PMU Image Length	PMU firmware original image length in bytes. (0-128KB).  If size > 0, PMUFW is prefixed to FSBL. If size = 0, no PMUFW image.
0x38	Total PMU FW Length	Total PMUFW image length in bytes.(PMUFW length + encryption overhead)

Table 9: Zynq UltraScale+ MPSoC Device Boot Header (cont'd)

Address Offset	Parameter	Description
0x3C	FSBL Image Length	Original FSBL image length in bytes. (0-250KB). If 0, XIP bootimage is assumed.
0x40	Total FSBL Length	FSBL image length + Encryption overhead of FSBL image + Auth. Cert., + 64byte alignment + hash size (Integrity check).
0x44	FSBL Image Attributes	See <a href="#">Bit Attributes</a> .
0x48	Boot Header Checksum	Sum of words from offset 0x20 to 0x44 inclusive. The words are assumed to be little endian.
0x4C-0x68	Obfuscated/Black Key Storage	Stores the Obfuscated key or Black key.
0x6C	Shutter Value	32-bit <code>PUF_SHUT</code> register value to configure PUF for shutter offset time and shutter open time.
0x70 -0x94	User-Defined Fields (UDF)	40 bytes.
0x98	Image Header Table Offset	Pointer to Image Header Table.
0x9C	Partition Header Table Offset	Pointer to Partition Header.
0xA0-0xA8	Secure Header IV	IV for secure header of bootloader partition.
0x0AC-0xB4	Obfuscated/Black Key IV	IV for Obfuscated or Black key.

## Zynq UltraScale+ MPSoC Boot Header Attribute Bits

Table 10: Zynq UltraScale+ MPSoC Boot Header Attribute Bits

Field Name	Bit Offset	Width	Default	Description
Reserved	31:16	16	0x0	Reserved. Must be 0.
BHDR RSA	15:14	2	0x0	0x3: RSA Authentication of the boot image will be done, excluding verification of PPK hash and SPK ID. All Others : RSA Authentication will be decided based on eFuse RSA bits.
Reserved	13:12	2	0x0	NA
CPU Select	11:10	2	0x0	0x0: R5 Single 0x1: A53 Single 32-bit 0x2: A53 Single 64-bit 0x3: R5 Dual
Hashing Select	9:8	2	0x0	0x0, 0x1 : No Integrity check 0x3: SHA3 for BI integrity check

Table 10: Zynq UltraScale+ MPSoC Boot Header Attribute Bits (cont'd)

Field Name	Bit Offset	Width	Default	Description
PUF-HD	7:6	2	0x0	0x3: PUF HD is part of boot header. All other: PUF HD is in eFuse
Reserved	5:0	6	0x0	Reserved for future use. Must be 0.

## 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 [Initialization Pairs and INT File Attribute](#).

Table 11: Zynq UltraScale+ MPSoC Register Initialization Table

Address Offset	Parameter	Description
0xB8 to 0xB4	Register Initialization Pairs: <address>:<value>: (2048 bytes)	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.

## 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

Address Offset	Parameter	Description
0x8B8 to 0xEC0	PUF Helper Data (1544 bytes)	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.

## 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**

Address Offset	Parameter	Description
0x00	Version	0x01010000 0x01020000 - 0x10 field is added
0x04	Count of Image Header	Indicates the number of image headers.
0x08	First Partition Header Offset	Pointer to first partition header (word offset).
0x0C	First Image Offset Header	Pointer to first image header (word offset).
0x10	Header Authentication Certificate	Pointer to header authentication certificate (word offset).
0x14	Secondary Boot Device	Options are:  0 - Same boot device 1 - QSPI-32 2 - QSPI-24 3 - NAND 4 - SD0 5 - SD1 6 - SDLS 7 - MMC 8 - USB 9 - ETHERNET 10 - PCIE 11 - SATA
0x18- 0x38	Padding	Reserved (0x0)
0x3C	Checksum	A sum of all the previous words in the image header.



## 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 also contains the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the AMD Zynq™ UltraScale+™ MPSoC.

**Table 14: Zynq UltraScale+ MPSoC Device Image Header**

Address Offset	Parameter	Description
0x00	Next image header offset	Link to next Image Header. 0 if last Image Header (word offset).
0x04	Corresponding partition header	Link to first associated Partition Header (word offset).
0x08	Reserved	Always 0.
0x0C	Partition Count	Value of the actual partition count.
0x10 - N	Image Name	Packed in big endian order. To reconstruct the string, unpack 4 bytes at a time, reverse the order, and concatenated. For example, the string "FSBL10.ELF" is packed as 0x10: 'L', 'B', 'S', 'F', 0x14: 'E', '.', '0', '1', 0x18: '\0', '\0', 'F', 'L'. The packed image name is a multiple of 4 bytes.
varies	String Terminator	0x00000
varies	Padding	Defaults to 0xFFFFFFFF to 64 bytes boundary.

## 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 AMD Zynq™ UltraScale+™ MPSoC.

**Table 15: Zynq UltraScale+ MPSoC Device Partition Header**

Offset	Name	Notes
0x0	Encrypted Partition Data Word Length	Encrypted partition data length.
0x04	Un-encrypted Data Word Length	Unencrypted data length.

Table 15: Zynq UltraScale+ MPSoC Device Partition Header (cont'd)

Offset	Name	Notes
0x08	Total Partition Word Length (Includes Authentication Certificate. See <a href="#">Authentication Certificate</a> ).	The total encrypted + padding + expansion + authentication length.
0x0C	Next Partition Header Offset	Location of next partition header (word offset).
0x10	Destination Execution Address <sub>LO</sub>	The lower 32-bits of executable address of this partition after loading.
0x14	Destination Execution Address <sub>HI</sub>	The higher 32-bits of executable address of this partition after loading.
0x18	Destination Load Address <sub>LO</sub>	The lower 32-bits of RAM address into which this partition is to be loaded.
0x1C	Destination Load Address <sub>HI</sub>	The higher 32-bits of RAM address into which this partition is to be loaded.
0x20	Actual Partition Word Offset	The position of the partition data relative to the start of the boot image. (word offset)
0x24	Attributes	See <a href="#">Zynq UltraScale+ MPSoC Partition Attribute Bits</a>
0x28	Section Count	The number of sections associated with this partition.
0x2C	Checksum Word Offset	The location of the checksum table in the boot image. (word offset)
0x30	Image Header Word Offset	The location of the corresponding image header in the boot image. (word offset)
0x34	AC Offset	The location of the corresponding Authentication Certificate in the boot image, if present (word offset)
0x38	Partition Number/ID	Partition ID.
0x3C	Header Checksum	A sum of the previous words in the Partition Header.

## Zynq UltraScale+ MPSoC Partition Attribute Bits

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

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

Bit Offset	Field Name	Description
31:24	Reserved	
23	Vector Location	Location of exception vector.  0: LOVEC (default) 1: HIVEC
22:20	Reserved	
19	Early Handoff	Handoff immediately after loading:  0: No Early Handoff 1: Early Handoff Enabled

**Table 16: AMD Zynq™ UltraScale+™ MPSoC Device Partition Attribute Bits (cont'd)**

Bit Offset	Field Name	Description
18	Endianness	0: Little Endian 1: Big Endian
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	A5X Exec State	0: AARCH64 (default) 1: AARCH32
2:1	Exception Level	0: EL0 1: EL1 2: EL2 3: EL3

Table 16: AMD Zynq™ UltraScale+™ MPSoC Device Partition Attribute Bits (cont'd)

Bit Offset	Field Name	Description
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 needs authentication. The Header Table Authentication Certificate is appended at end of the content to the header tables.

The AMD 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

Authentication Certificate		
0x00	Authentication Header = 0x0101000. See <a href="#">Zynq UltraScale+ MPSoC Authentication Certification Header</a> .	
0x04	SPK ID	
0x08	UDF (56 bytes)	
0x40	PPK	Mod (512)
0x240		Mod Ext (512)
0x440		Exponent (4 bytes)
0x444		Pad (60 bytes)
0x480	SPK	Mod (512 bytes)
0x680		Mod Ext (512 bytes)
0x880		Exponent (4 bytes)
0x884		Pad (60 bytes)
0x8C0	SPK Signature = RSA-4096 ( PSK, Padding    SHA-384 (SPK + Authentication Header + SPK-ID))	
0xAC0	Boot Header Signature = RSA-4096 ( SSK, Padding    SHA-384 (Boot Header))	
0xCC0	Partition Signature = RSA-4096 ( SSK, Padding    SHA-384 (Partition    Padding    Authentication Header    SPK ID    UDF    PPK    SPK    SPK Signature    BH Signature))	

**Note:** FSBL Signature is calculated as follows:

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

## Zynq UltraScale+ MPSoC Authentication Certification Header

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

**Table 18: Authentication Header Bit Fields**

Bit Field	Description	Notes
31:20	Reserved	0
19:18	SPK/User eFuse Select	01: SPK eFuse 10: User eFuse
17:16	PPK Key Select	0: PPK0 1: PPK1
15:14	Authentication Certificate Format	00: PKCS #1 v1.5
13:12	Authentication Certificate Version	00: Current AC
11	PPK Key Type	0: Hash Key
10:9	PPK Key Source	0: eFUSE
8	SPK Enable	1: SPK Enable
7:4	Public Strength	0 : 2048b 1 : 4096 2:3 : Reserved
3:2	Hash Algorithm	1: SHA3/384 2:3 Reserved
1:0	Public Algorithm	0: Reserved 1: RSA 2: Reserved 3: Reserved

## 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+ MPSoC secure header is shown in the following table.

Figure 4: Zynq UltraScale+ MPSoC Secure Header

AES

	Partition#0 (FSBL)				Partition#1				Partition#2			
	Encrypted Using		Contents		Encrypted Using		Contents		Encrypted Using		Contents	
Secure Header	Key0	IV0	-	IV1_#0	Key0	IV0+0x01	Key1_#1	IV1_#1	Key0	IV0+0x02	Key1_#2	IV1_#2
Block #0	Key0	IV1_#0	-	-	Key1_#1	IV1_#1	-	-	Key1_#2	IV1_#2	-	-

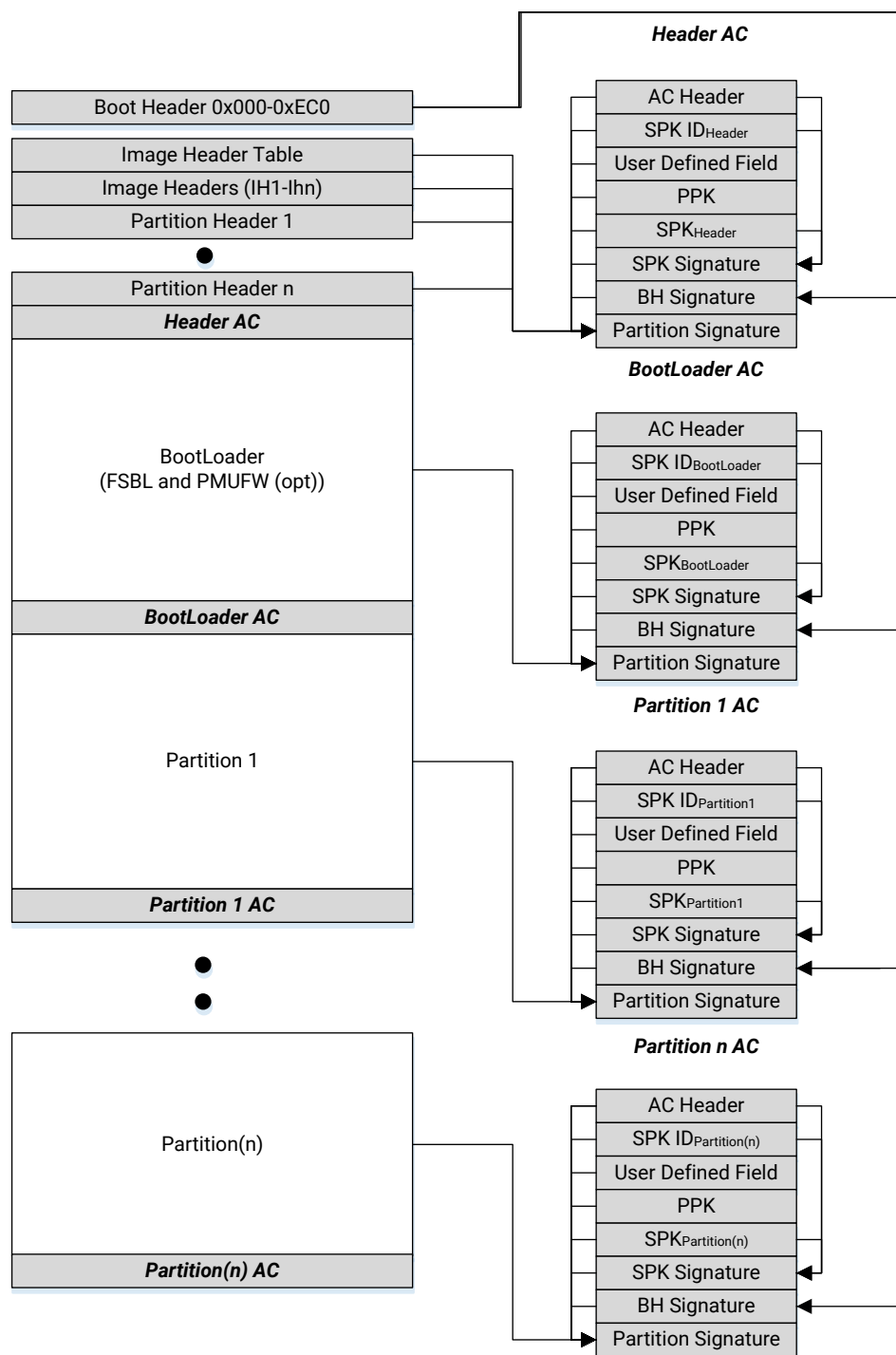
AES with Key rolling

	Partition#0 (FSBL)				Partition#1				Partition#2			
	Encrypted Using		Contents		Encrypted Using		Contents		Encrypted Using		Contents	
Secure Header	Key0	IV0	-	IV1_#0	Key0	IV0+0x01	Key1_#1	IV1_#1	Key0	IV0+0x02	Key1_#2	IV1_#2
Block #0	Key0	IV1_#0	Key2_#0	IV2_#0	Key1_#1	IV1_#1	Key2_#1	IV2_#1	Key1_#2	IV1_#2	Key2_#2	IV2_#2
Block #1	Key2_#0	IV2_#0	Key3_#0	IV3_#0	Key2_#1	IV2_#2	Key3_#1	IV3_#1	Key2_#2	IV2_#2	Key3_#2	IV3_#2
Block #2	Key3_#0	IV3_#0	Key4_#0	IV4_#0	Key3_#1	IV3_#2	Key4_#1	IV4_#1	Key3_#2	IV3_#2	Key4_#2	IV4_#2
...	...	...	...	...	...	...	...	...	...	...	...	...

## Zynq UltraScale+ MPSoC Boot Image Block Diagram

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

Figure 5: Zynq UltraScale+ MPSoC Device Boot Image Block Diagram



X18916-081518

---

# Versal Adaptive SoC Boot Image Format

The following is a diagram of the components that can be included in an AMD Versal™ adaptive SoC boot image called programmable device image (PDI).

## Platform Management Controller

The platform management controller (PMC) in Versal adaptive SoC is responsible for platform management of the Versal adaptive SoC, including boot and configuration. This chapter is focused on the boot image format processed by the two PMC MicroBlaze processors, the ROM code unit (RCU), and the platform processing unit (PPU):

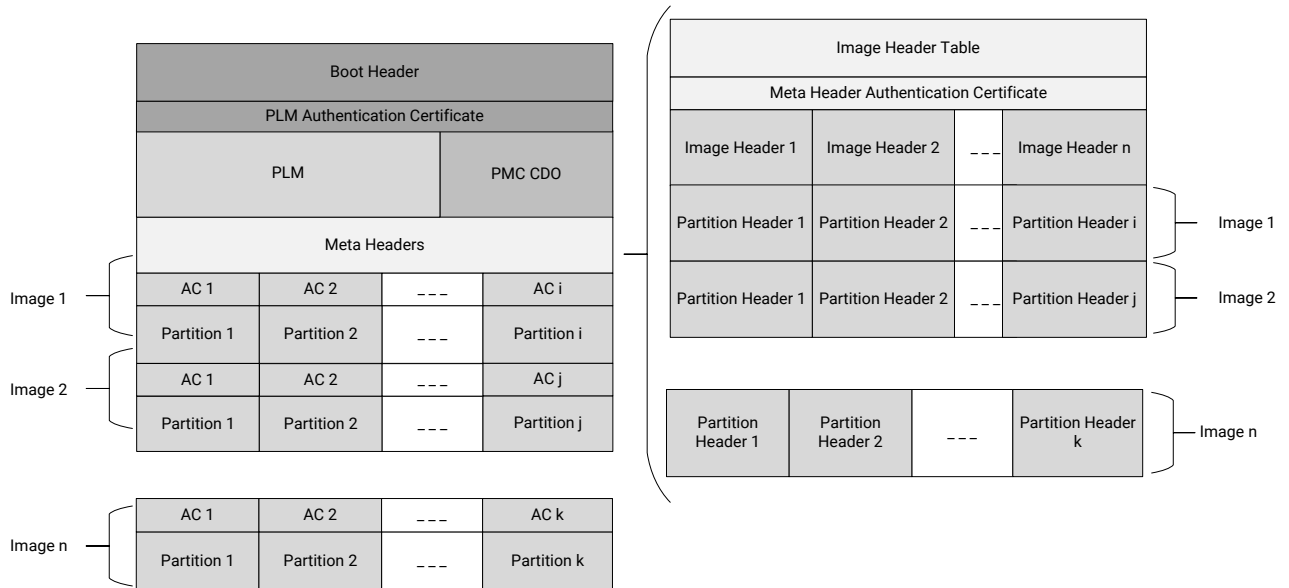
- **RCU:** The ROM code unit contains a triple-redundant MicroBlaze processor and read-only memory (ROM) which contains the executable BootROM. The BootROM executable is metal-masked and unchangeable. The MicroBlaze processor in the RCU is responsible for validating and running the BootROM executable. The RCU is also responsible for post-boot security monitoring and physical unclonable function (PUF) management.
- **PPU:** The platform processing unit contains a triple-redundant MicroBlaze processor and 384 KB of dedicated PPU RAM. The MicroBlaze in the PPU is responsible for running the platform loader and manager (PLM).

In Versal adaptive SoC, the adaptable engine (PL) consists of rCDO and rNPI files. The rCDO file mainly contains CFrame data along with PL and NoC power domain initialization commands. The rNPI file contains configuration data related to the NPI blocks. NPI blocks include NoC elements: NMU, NSU, NPS, NCRB; DDR, XPHY, XPIO, GTY, MMCMs, and so on.

**Note:** AMD Versal™ adaptive SoC includes SSI technology devices. For more information, see [Chapter 6: SSIT Support](#).



Figure 6: Versal Adaptive SoC Boot Image Block Diagram



X22829-030722

## Versal Adaptive SoC Boot Header

Boot header is used by PMC BootROM. Based on the attributes set in the boot header, PMC BootROM validates the Platform Loader and Manager (PLM) and loads it to the PPU RAM. The first 16 bytes are intended for SelectMAP Bus detection. PMC BootROM and PLM ignore this data so Bootgen does not include this data in any of its operations like checksum/SHA/RSA/Encryption and so on. The following code snippet is an example of SelectMAP Bus width detection pattern bits. Bootgen places the following data in first 16-bytes as per the width selected.

The individual image header width and the corresponding bits are shown in the following list:

- **X8:** [LSB] 00 00 00 DD 11 22 33 44 55 66 77 88 99 AA BB CC [MSB]
- **X16:** [LSB] 00 00 DD 00 22 11 44 33 66 55 88 77 AA 99 CC BB [MSB]
- **X32:** [LSB] DD 00 00 00 44 33 22 11 88 77 66 55 CC BB AA 99 [MSB]

**Note:** The default SelectMAP width is X32.

The following table shows the boot header format for an AMD Versal™ adaptive SoC.

Table 19: Versal Adaptive SoC Boot Header Format

Offset (Hex)	Size (Bytes)	Description	Details
0x00	16	SelectMAP bus width	Used to determine if the SelectMAP bus width is x8, x16, or x32
0x10	4	QSPI bus width	QSPI bus width description. This is required to identify the QSPI flash in single/dual stacked or dual parallel mode. 0xAA995566 in the little endian format.
0x14	4	Image identification	Boot image identification string. Contains 4 bytes X, N, L, X in byte order, which is 0x584c4e58 in the little endian format.
0x18	4	Encryption key source	This field is used to identify the AES key source: 0x00000000 - Unencrypted 0xA5C3C5A3 - eFUSE red key 0xA5C3C5A5 - eFUSE black key 0x3A5C3C5A - BBRAM red key 0x3A5C3C59 - BBRAM black key 0xA35C7C53 - Boot Header black key
0x1C	4	PLM source offset	PLM source start address in PDI
0x20	4	PMC data load address	PMC CDO address to load
0x24	4	PMC data length	PMC CDO length
0x28	4	Total PMC data length	PMC CDO length including authentication and encryption overhead
0x2C	4	PLM length	PLM original image size
0x30	4	Total PLM length	PLM image size including the authentication and encryption overhead
0x34	4	Boot header attributes	<a href="#">Versal Adaptive SoC Boot Header Attributes</a>
0x38	32	Black key	256-bit key, only valid when encryption status is set to black key in boot header
0x58	12	Black IV	Initialization vector used when decrypting the black key
0x64	12	Secure header IV	Secure header initialization vector

Table 19: Versal Adaptive SoC Boot Header Format (cont'd)

Offset (Hex)	Size (Bytes)	Description	Details
0x70	4	PUF shutter value	Length of the time the PUF samples before it closes the shutter  <b>Note:</b> This shutter value must match the shutter value that was used during PUF registration.
0x74	12	Secure Header IV for PMC Data	The IV used to decrypt secure header of PMC data.
0x80	68	Reserved	Populate with zeroes.
0xC4	4	Meta Header Offset	Offset to the start of meta header.
0xC8 - 0x124	96	Reserved	
0x128	2048	Register init	Stores register write pairs for system register initialization
0x928	1544	PUF helper data	PUF helper data
0xF30	4	Checksum	Header checksum
0xF34	76	SHA3 padding	SHA3 standard padding

## Versal Adaptive SoC Boot Header Attributes

The image attributes are described in the following table.

Table 20: Versal Adaptive SoC Boot Header Attributes

Field Name	Bit Offset	Width	Default Value	Description
Reserved	[31:18]	14	0x0	Reserved for future use, Must be 0
PUF Mode	[17:16]	2	0x0	0x3 - PUF 4K mode. 0x0 - PUF 12K mode.
Boot Header Authentication	[15:14]	2	0x0	0x3 - Authentication of the boot image is done, excluding verification of PPK hash and SPK ID. All others - Authentication is decided based on eFUSE RSA/ECDSA bits.
Reserved	[13:12]	2	0x0	Reserved for future use, Must be 0
DPA counter measure	[11:10]	2	0x0	0x3 - Enabled All others disabled. (eFUSE over rides this)

Table 20: Versal Adaptive SoC Boot Header Attributes (cont'd)

Field Name	Bit Offset	Width	Default Value	Description
Checksum selection	[9:8]	2	0x0	0x0, 0x1, 0x2 - Reserved 0x3 - SHA3 is used as hash function to do Checksum.
PUF HD	[7:6]	2	0x0	0x3 - PUF HD is part of boot header All other - PUF HD is in eFUSE.
Reserved	[5:0]	6	0x0	Reserved

## Versal Adaptive SoC Image Header Table

The following table contains generic information related to the PDI image.

Table 21: Versal Adaptive SoC Image Header Table

Offset	Name	Description
0x0	Version	0x00040000(v4.0) : <ol style="list-style-type: none"> <li>Added AAD support for IHT.</li> <li>Hash is included into the 32k secure chunk.</li> </ol> 0x00030000(v3.0): updated secure chunk size to 32 KB from 64 KB 0x00020000(v2.00): IHT, PHT sizes doubled
0x4	Total Number of Images	Total number of images in the PDI
0x8	Image Header Offset (Word)	Address to start of first image header
0xC	Total Number of Partitions	Total number of partitions in the PDI
0x10	Partition Header Offset (Word)	Offset to the start of partitions headers
0x14	Secondary boot device address	Indicates the address where secondary image is present. This is only valid if secondary boot device is present in attributes
0x1C	Image Header Table Attributes	Refer to <a href="#">Table 22: Versal Adaptive SoC Image Header Table Attributes</a>
0x20	PDI ID	Used to identify a PDI
0x24	Parent ID	ID of initial boot PDI. For boot PDI, it is same as the PDI ID
0x28	Identification string	Full PDI if present with boot header - "FPDI" Partial/Sub-system PDI - "PPDI"

Table 21: Versal Adaptive SoC Image Header Table (cont'd)

Offset	Name	Description
0x2C	Headers size	0-7: Image header table size in words 8-15: Image header size in words 16-23: Partition header size in words 24-31: Reserved
0x30	Total meta header length (Word)	Including authentication and encryption overhead (excluding IHT and including AC)
0x34 -0x3C	IV for encryption of meta header	IV for decrypting SH of header table
0x40	Encryption status	Encryption key source, only key source used for PLM is valid for meta header. 0x00000000 - Unencrypted 0xA5C3C5A3 - eFuse red key 0xA5C3C5A5 - eFUSE black key 0x3A5C3C5A - BBRAM red key 0x3A5C3C59 - BBRAM black key 0xA35C7C53 - Boot Header black key
0x48	Meta Header AC Offset (Word)	Word Offset to Meta Header Authentication Certificate
0x4c	Meta Header Black/IV	IV that is used to encrypt the Black key used to encrypt the Meta Header.
0x58	Optional Data Length (Word)	Size of Optional Data available in Bootloader
0x5C - 0x78	Reserved	0x0
0x7C	Checksum	A sum of all the previous words in the image header table

### Image Header Table Attributes

The image header tables are described in the following table.

Table 22: Versal Adaptive SoC Image Header Table Attributes

Bit Field	Name	Description
31:14	Reserved	0
14	PUF Helper Data Location	Location of the PUF Helper Data efuse/BH
12	dpacm enable	DPA Counter Measure enable or not

Table 22: Versal Adaptive SoC Image Header Table Attributes (cont'd)

Bit Field	Name	Description
11:6	Secondary boot device	<p>Indicates the device on which rest of the data is present in.</p> <p>0 - Same boot device (default)</p> <p>1 - QSPI32</p> <p>2 - QSPI24</p> <p>3 - NAND</p> <p>4 - SD0</p> <p>5 - SD1</p> <p>6 - SDLS</p> <p>7 - MMC</p> <p>8 - USB</p> <p>9 - ETHERNET</p> <p>10 - PCIe</p> <p>11 - SATA</p> <p>12 - OSPI</p> <p>13 - SMAP</p> <p>14 - SBI</p> <p>15 - SD0RAW</p> <p>16 - SD1RAW</p> <p>17 - SDLSRAW</p> <p>18 - MMCRAW</p> <p>19 - MMC0</p> <p>20 - MMC0RAW</p> <p>21 - imagestore</p> <p>All others are reserved</p> <p><b>Note:</b> These options are supported for various devices in Bootgen. For the exact list of secondary boot devices supported by any device, refer to its corresponding Systems Software Developers Guide (SSDG).</p>
5:0		Reserved

## Versal Adaptive SoC Image Header

The image header is an array of structures containing information related to each image, such as an ELF file, CFrame, NPI, CDOs, 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. An image header points to the partitions (partition headers) that are associated with this image. Multiple partition files can be grouped within an image using the BIF keyword "image"; this is useful for combining all the partitions related to a common subsystem or function in a group. Bootgen creates the required partitions for each file and creates a common image header for that image. The following table contains the information of number of partitions related to an image.

Table 23: Versal Adaptive SoC Image Header

Offset	Name	Description
0x0	First Partition Header (Word)	Word offset to first partition header
0x4	Number of Partitions	Number of partitions present for this image
0x8	Revoke ID	Revoke ID for Meta Header
0xC	Image Attributes	See Image Attributes table
0x10-0x1C	Image Name	ASCII name of the image. Max of 16 characters. Fill with Zeros when padding is required.
0x20	Image/Node ID	Defines the resource node the image is initializing
0x24	Unique ID	Defines the affinity/compatibility identifier when required for a given device resource
0x28	Parent Unique ID	Defines the required parent resource UID for the configuration content of the image, if required
0x2c	Function ID	Identifier used to capture the unique function of the image configuration data
0x30	DDR Low Address for Image Copy	The DDR lower 32-bit address where the image must be copied when memcpy is enabled in BIF
0x34	DDR High Address for Image Copy	The DDR higher 32-bit address where image must be copied when memcpy is enabled in BIF
0x38	Reserved	
0x3C	Checksum	A sum of all the previous words.

The following table shows the Image Header Attributes.

Table 24: Versal Adaptive SoC Image Header Attributes

Bit Field	Name	Description
31:9	Reserved	0
8	Delay Hand off	0 – Handoff the image now (default) 1 – Handoff the image later
7	Delay load	0 – Load the image now (default) 1 – Load the image later
6	Copy to memory	0 – No copy to memory (Default) 1 – Image to be copied to memory
5:3	Image Owner	0 - PLM (default) 1 - Non-PLM 2-7 – Reserved
2:0	Reserved	0

## Versal Adaptive SoC Partition Header

The partition header contains details of the partition and is described in the table below.

**Table 25: Versal Adaptive SoC Partition Header Table**

Offset	Name	Description
0x0	Partition Data Word Length	Encrypted partition data length
0x4	Extracted Data Word Length	Unencrypted data length
0x8	Total Partition Word Length (Includes Authentication Certificate)	The total encrypted + padding + expansion + authentication length
0xC	Next Partition header offset (Word)	Offset of next partition header
0x10	Destination Execution Address (Lower Half)	The lower 32 bits of the executable address of this partition after loading.
0x14	Destination Execution Address (Higher Half)	The higher 32 bits of the executable address of this partition after loading.
0x18	Destination Load Address (Lower Half)	The lower 32 bits of the RAM address into which this partition is to be loaded. For elf files, Bootgen automatically reads from elf format. For RAW data, you have to specify where to load it. For CFI and configuration data it must be 0xFFFF_FFFF
0x1C	Destination Load Address (Higher Half)	The higher 32 bits of the RAM address into which this partition is to be loaded. For elf files, Bootgen automatically reads from elf format. For RAW data, you have to specify where to load it. For CFI and configuration data it must be 0xFFFF_FFFF
0x20	Data Word Offset in Image	The position of the partition data relative to the start of the boot image.
0x24	Attribute Bits	See Partition Attributes Table
0x28	Section Count	If image type is elf, it says how many more partitions are associated with this elf.
0x2C	Checksum Word Offset	The location of the checksum word in the boot image.
0x30	Partition ID	Partition ID
0x34	Authentication Certification Word Offset	The location of the Authentication Certification in the boot image.
0x38 – 0x40	IV	IV for the secure header of the partition.



Table 25: Versal Adaptive SoC Partition Header Table (cont'd)

Offset	Name	Description
0x44	Encryption Key select	Encryption status: 0x00000000 – Unencrypted 0xA5C3C5A3 – eFuse Red Key 0xA5C3C5A5 – eFuse Black Key 0x3A5C3C5A – BBRAM Red Key 0x3A5C3C59 – BBRAM Black Key 0xA35C7C53 – Boot Header Black Key 0xC5C3A5A3 – User Key 0 0xC3A5C5B3 – User Key 1 0xC5C3A5C3 – User Key 2 0xC3A5C5D3 – User Key 3 0xC5C3A5E3 – User Key 4 0xC3A5C5F3 – User Key 5 0xC5C3A563 – User Key 6 0xC3A5C573 – User Key 7 0x5C3CA5A3 – eFuse User Key 0 0x5C3CA5A5 – eFuse User Black Key 0 0xC3A5C5A3 – eFuse User Key 1 0xC3A5C5A5 – eFuse User Black Key 1
0x48	Black IV	IV used for encrypting the key source of that partition.
0x54	Revoke ID	Partition revoke ID
0x58-0x78	Reserved	0
0x7C	Header Checksum	A sum of the previous words in the Partition Header

The following table lists the partition header table attributes.

Table 26: Versal Adaptive SoC Partition Header Table Attributes

Bit Field	Name	Description
31:29	Reserved	0x0
28:27	DPA CM Enable	0 – Disabled 1 – Enabled
26:24	Partition Type	0 – Reserved 1 – elf 2 – Configuration Data Object 3 – Cframe Data (PL data) 4 – Raw Data 5 – Raw elf 6 – CFI GSR CSC unmask frames 7 – CFI GSR CSC mask frames

Table 26: Versal Adaptive SoC Partition Header Table Attributes (cont'd)

Bit Field	Name	Description
23	HiVec	VInitHi setting for RPU/APU(32-bit) processor 0 – LoVec 1 – HiVec
22:19	Reserved	0
18	Endianness	0 – Little Endian (Default) 1 – Big Endian
17:16	Partition Owner	0 - PLM (Default) 1 - Non-PLM 2,3 – Reserved
15:14	PUF HD location	0 - eFuse 1 - Boot header
13:12	Checksum Type	000b - No Checksum(Default) 011b - SHA3
11:8	Destination CPU	0 – None (Default for non-elf files) 1 - A72-0 2 - A72-1 3 - Reserved 4 - Reserved 5 - R5-0 6 - R5-1 7- R5-L 8 – PSM 9 - AIE 10-15 – Reserved
4:7	Reserved	0x0
3	A72 CPU execution state	0 - Aarch64 (default) 1 - Aarch32
2:1	Exception level (EL) that the A72 core must be configured for	00b – EL0 01b – EL1 10b – EL2 11b – EL3 (Default)
0	TZ secure partition	0 – Non-Secure (Default) 1 – Secure This bit indicates if the core that the PLM needs to configure (on which this partition needs to execute) must be configured as TrustZone secure or not. By default, this must be 0.

## Versal Adaptive SoC 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/PLM 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. Unlike the other devices, the Authentication Certificate is prepended or attached to the beginning of the actual partition, for which authentication is enabled. If you want Bootgen to perform authentication on the meta headers, specify it explicitly under the 'metaheader' bif attribute. See [Chapter 9: BIF Attribute Reference](#) for information on usage.

Versal adaptive SoC uses RSA-4096 authentication and ECDSA algorithms for authentication. The following table provides the format of the Authentication Certificate for the Versal adaptive SoC.

**Table 27: Versal Adaptive SoC Authentication Certificate – ECDSA p384**

Authentication Certificate Bits		Description
0x00	Authentication Header. See <a href="#">Versal Adaptive SoC Authentication Certification Header</a>	
0x04	Revoke ID	
0x08	UDF (56 bytes)	
0x40	PPK	x (48 bytes)
		y (48 bytes)
		Pad 0x00 (932 bytes)
0x444	PPK SHA3 Pad (12 bytes)	
0x450	SPK	x (48 bytes)
		y (48 bytes)
		Pad 0x00 (932 bytes)
0x854	SPK SHA3 Pad (4 bytes)	
0x858	Alignment (8 bytes)	
0x860	SPK Signature(r+s+pad)(48+48+416)	
0xA60	BH/IHT Signature(r+s+pad)(48+48+416)	
0xC60	Partition Signature(r+s+pad)(48+48+416)	

**Table 28: Versal Adaptive SoC Authentication Certificate – ECDSA p521**

Authentication Certificate Bits		Description
0x00	Authentication Header. See <a href="#">Versal Adaptive SoC Authentication Certification Header</a>	
0x04	Revoke ID	
0x08	UDF (56 bytes)	

Table 28: Versal Adaptive SoC Authentication Certificate – ECDSA p521 (cont'd)

Authentication Certificate Bits		Description
0x40	PPK	PPK x (66 bytes)
		y (66 bytes)
		Pad 0x00 (896 bytes)
0x444	PPK SHA3 Pad (12 bytes)	
0x450	SPK	SPK x (66 bytes)
		y (66 bytes)
		Pad 0x00 (896 bytes)
0x854	SPK SHA3 Pad (4 bytes)	
0x858	Alignment (8 bytes)	
0x860	SPK Signature(r+s+pad)(66+66+380)	
0xA60	BH/IHT Signature(r+s+pad)(66+66+380)	
0xC60	Partition Signature(r+s+pad)(66+66+380)	

Table 29: Versal Adaptive SoC Authentication Certificate – RSA

Authentication Certificate Bits		Description
0x00	Authentication Header. See <a href="#">Versal Adaptive SoC Authentication Certification Header</a>	
0x04	Revoke ID	
0x08	UDF (56 bytes)	
0x40	PPK	Mod (512 bytes)
		Mod Ext (512 bytes)
		Exponent (4 bytes)
0x444	PPK SHA3 Pad (12 bytes)	
0x450	SPK	Mod (512 bytes)
		Mod Ext (512 bytes)
		Exponent (4 bytes)
0x854	SPK SHA3 Pad (4 bytes)	
0x858	Alignment (8 bytes)	
0x860	SPK Signature	
0xA60	BH/IHT Signature	
0xC60	Partition Signature	

## Versal Adaptive SoC Authentication Certification Header

The following table describes the Authentication Header bit fields for the Versal adaptive SoC.

Table 30: Authentication Header Bit Fields

Bit Fields	Description	Notes
31:16	Reserved	0
15-14	Authentication Certificate Format	00 -RSAPSS
13-12	Authentication Certificate Version	00: Current AC
11	PPK Key Type	0: Hash Key
10-9	PPK Key Source	0: eFUSE
8	SPK Enable	1: SPK Enable
7-4	Public Strength	0 - ECDSA p384 1 - RSA 4096 2 - ECDSA p521
3-2	Hash Algorithm	1-SHA3
1-0	Public Algorithm	1-RSA 2-ECDSA

**Note:**

1. For the Bootloader partition:
  - a. The offset 0xA60 of the AC holds the Boot Header Signature.
  - b. The offset 0xC60 of the AC holds the signature of PLM and PMCDATA.
2. For the Header tables:
  - a. The offset 0xA60 of the AC holds the IHT Signature.
  - b. The offset 0xC60 of the AC holds the signature of all the headers except IHT.
3. For any other partition:
  - a. The offset 0xA60 of the AC is zeroized.
  - b. The offset 0xC60 of the AC holds the signature of that partition.

# Creating Boot Images

---

## Boot Image Format (BIF)

The AMD 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
- Bootloader
  - First stage bootloader (FSBL) for AMD Zynq™ devices and AMD Zynq™ UltraScale+™ MPSoCs
  - Platform loader and manager (PLM) for AMD Versal™ adaptive SoC
  - **Note:** It is recommended to use the same release version of bootloader (FSBL/PLM) and Bootgen together.
- 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, an AMD Zynq™ 7000 SoC device, AMD Versal™ adaptive SoC series, or an AMD Zynq™ UltraScale+™ MPSoC device, you must provide the appropriate [arch](#) command option to Bootgen. The following appendices list and describe the available options to direct Bootgen behavior.

- [Chapter 8: Use Cases and Examples](#)
- [Chapter 9: BIF Attribute Reference](#)
- [Chapter 10: Command Reference](#)

The format of the boot image conforms to a hybrid mix of hardware and software requirements. The boot header is required by the BootROM loader which loads a single partition, typically the bootloader. The remainder of the boot image is loaded and processed by the bootloader. Bootgen generates a boot image by combining a list of partitions. These partitions can be:

- FSBL or PLM
- Secondary Stage Boot Loader (SSBL) like U-Boot
- Bitstream PL CFrame data, `.rcdo`, and `.rnp1`
- Linux
- Software applications to run on processors
- User data
- Boot image generated by Bootgen. This is useful for appending new partitions to a previously generated boot image.

**Note:** Do avoid mix and match of tools release and initial PDI artifacts like PLM.elf, PSM.elf PMC/LPD/FPD.cdo from another tools release.

## 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. For example, if an ELF file has multiple loadable sections that are non-contiguous, then each section can be a separate partition. BIF file syntax takes the following form:

```
new_bif:
{
    id = 0x2
    id_code = 0x04ca8093
    extended_id_code = 0x01
    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition
        {
            id = 0x11, type = bootloader,
            file = /path/to/plm.elf
        }
        partition
        {
            type = pmcdata, load = 0xf2000000,
            file = /path/to/pmc_cdo.bin
        }
    }
}
```

**Note:** The above format is for AMD Versal™ devices only.

```
<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.
- 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:

```
<bootimage_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 31: Bootgen Supported Files**

Device Supported	Extension	Description	Notes
Supported by all devices	.bin	Binary	Raw binary file.
	.dtb	Binary	Raw binary file.
	image.gz	Binary	Raw binary file.
	.elf	Executable Linked File (ELF)	Symbols and headers removed.
	.int	Register initialization file	
	.nky	AES key	
	.pub/.pem	RSA key	
	.sig	Signature files	Signature files generated by Bootgen or HSM.
Versal	.redo	CFI Files	For Versal devices only.
	.cdo/.npi/ .rnpi	CDO files	Configuration Data Object files. For Versal devices only.
	.bin/.pdi	Boot image	Boot image generated using Bootgen.

Table 31: Bootgen Supported Files (cont'd)

Device Supported	Extension	Description	Notes
Zynq 7000/Zynq UltraScale+ MPSoC/FPGA	.bit/.rbt	Bitstream	Strips the BIT file header.

## BIF Syntax for Versal Adaptive SoC

The following example shows the detailed manner in which you can write a BIF while grouping the partitions together. The BIF syntax has changed for Versal adaptive SoC to support the concept of subsystems, where multiple partitions can be combined to together to form an image, also called as subsystem with one image header.

**Note:** The partitions under the same image {} block is merged to form a single subsystem.

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0x01
            type = bootloader
            file = gen_files/plm.elf
        }
        partition
        {
            id = 0x09
            type = pmcdata, load = 0xf2000000
            file = gen_files/pmc_data.cdo
        }
    }
    image
    {
        name = lpd
        id = 0x4210002
        partition
        {
            id = 0x0C
            type = cdo
            file = gen_files/lpd_data.cdo
        }
        partition
        {
            id = 0x0B
            core = psm
            file = static_files/psm_fw.elf
        }
    }
    image
    {
        name = pl_cfi
        id = 0x18700000
    }
}
```

```

        partition
        {
            id = 0x03
            type = cdo
            file = system.rcdo
        }
        partition
        {
            id = 0x05
            type = cdo
            file = system.rnpi
        }
    }
    image
    {
        name = fpd
        id = 0x420c003
        partition
        {
            id = 0x08
            type = cdo
            file = gen_files/fpd_data.cdo
        }
    }
}

```

The following example shows how you can write a BIF in a concise manner by grouping the partitions together.

```

new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { id = 0x01, type = bootloader, file = gen_files/plm.elf }
        { id = 0x09, type = pmcdata, load = 0xf2000000, file = gen_files/
pmc_data.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        { id = 0x0C, type = cdo, file = gen_files/lpd_data.cdo }
        { id = 0x0B, core = psm, file = static_files/psm_fw.elf }
    }
    image
    {
        name = pl_cfi, id = 0x18700000
        { id = 0x03, type = cdo, file = system.rcdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = gen_files/fpd_data.cdo }
    }
}

```

## Attributes

The following table lists the Bootgen attributes. Each attribute has a link to a longer description in the left column with a short description in the right column. The architecture name indicates which AMD devices uses that attribute:

- `zynq`: Zynq 7000 SoC device
- `zynqmp`: AMD Zynq™ UltraScale+™ MPSoC device
- `fpga`: Any 7 series and above devices
- `versal`: AMD Versal™ adaptive SoC.

For more information, see [Chapter 9: BIF Attribute Reference](#).

**Table 32: Bootgen Attributes and Description**

Option/Attribute	Description	Used By
<a href="#">aarch32_mode</a>	Specifies the binary file that is to be executed in 32-bit mode.	<ul style="list-style-type: none"> <li>• <code>zynqmp</code></li> <li>• <code>versal</code></li> </ul>
<a href="#">aeskeyfile</a> <aes_key_filepath>	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 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. For example: If encryption is selected for bitstream in the BIF file, the output is an encrypted bitstream.	<ul style="list-style-type: none"> <li>• All</li> </ul>
<a href="#">alignment</a> <byte>	Sets the byte alignment. The partition is padded to be aligned to a multiple of this value. This attribute cannot be used with offset.	<ul style="list-style-type: none"> <li>• <code>zynq</code></li> <li>• <code>zynqmp</code></li> </ul>
<a href="#">auth_params</a> <options>	Extra options for authentication: <ul style="list-style-type: none"> <li>• <code>ppk_select</code>: 0=1, 1=2 of two PPKs supported.</li> <li>• <code>spk_id</code>: 32-bit ID to differentiate SPKs.</li> <li>• <code>spk_select</code>: To differentiate spk and user efuses. Default is spk-efuse.</li> <li>• <code>header_auth</code>: To authenticate headers when no partition is authenticated.</li> </ul>	<ul style="list-style-type: none"> <li>• <code>zynqmp</code></li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">authentication</a> <option>	<p>Specifies the partition to be authenticated.</p> <ul style="list-style-type: none"> <li>Authentication for Zynq is done using RSA-2048.</li> <li>Authentication for Zynq UltraScale+ MPSoCs is done using RSA-4096.</li> <li>Authentication for Versal adaptive Soc is done using RSA-4096, ECDSA-p384, and ECDSA-p521.</li> </ul> <p>The arguments are:</p> <ul style="list-style-type: none"> <li><code>none</code>: Partition not signed.</li> <li><code>ecdsa-p384</code>: partition signed using ecdsa-p384 curve</li> <li><code>ecdsa-p521</code>: partition signed using ecdsa-p521 curve</li> <li><code>rsa</code>: Partition signed using RSA algorithm.</li> </ul>	<ul style="list-style-type: none"> <li>All</li> </ul>
<a href="#">bbram_kek_iv</a> <filename>	Specifies the IV that is used to encrypt the corresponding key. <code>bbram_kek_iv</code> is valid with <code>keysrc=bbram_blk_key</code> .	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">bh_kek_iv</a> <filename>	Specifies the IV that is used to encrypt the corresponding key. <code>bh_kek_iv</code> is valid with <code>keysrc=bh_blk_key</code> .	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">bh_key_iv</a> <filename>	Initialization vector used when decrypting the obfuscated key or a black key.	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<a href="#">bh_keyfile</a> <filename>	<p>256-bit obfuscated key or black key to be stored in the Boot Header. This is only valid when <code>keysrc</code> for encryption is <code>bh_gry_key</code> or <code>bh_blk_key</code>.</p> <p><b>Note:</b> Obfuscated key is not supported for Versal devices.</p>	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">bhsignature</a> <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 <code>boothheader.sha384.sig</code> .	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">big_endian</a>	Specifies the binary file is in big endian format.	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">blocks</a> <block sizes>	Specifies 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.	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">boot_config</a> <options>	This attribute specifies the parameters that are used to configure the boot image.	<ul style="list-style-type: none"> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>boot_device</code> <options>	<p>Specifies the secondary boot device. Indicates the device on which the partition is present. Options are:</p> <ul style="list-style-type: none"> <li><code>qspi32</code></li> <li><code>qspi24</code></li> <li><code>nand</code></li> <li><code>sd0</code></li> <li><code>sd1</code></li> <li><code>sd-ls</code></li> <li><code>mmc</code></li> <li><code>usb</code></li> <li><code>ethernet</code></li> <li><code>pcie</code></li> <li><code>sata</code></li> <li><code>ospi</code></li> <li><code>smap</code></li> <li><code>sbi</code></li> <li><code>sd0-raw</code></li> <li><code>sd1-raw</code></li> <li><code>sd-ls-raw</code></li> <li><code>mmc-raw</code></li> <li><code>mmc0</code></li> <li><code>mmc0-raw</code></li> </ul> <p><b>Note:</b> These options are supported for various devices in Bootgen. For a list of secondary boot options, see the <i>Versal Adaptive SoC System Software Developers Guide</i> (UG1304) or the <i>Zynq UltraScale+ MPSoC: Software Developers Guide</i> (UG1137). For hardware/register/interface information and primary boot modes, refer to the corresponding TRM, such as the <i>Zynq UltraScale+ Device Technical Reference Manual</i> (UG1085), the <i>Versal Adaptive SoC Technical Reference Manual</i> (AM011), or the <i>Versal Adaptive SoC Register Reference</i> (AM012).</p>	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<code>bootimage</code> <filename.bin>	<p>Specifies that the listed input file is a boot image that was created by Bootgen.</p>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">bootloader</a> <partition>	Specifies the partition is a bootloader (FSBL/PLM). This attribute is specified along with other partition BIF attributes.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">bootvectors</a> <vector_values>	Specifies the vector table for execute in place (XIP).	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<a href="#">checksum</a> <options>	<p>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:</p> <ul style="list-style-type: none"> <li><code>none</code>: No checksum operation.</li> <li><code>md5</code>: For AMD Zynq™ 7000 SoC devices only</li> <li><code>sha3</code>: For AMD Zynq™ UltraScale+™ MPSoC and Versal devices.</li> </ul> <p><b>Note:</b> Zynq devices do not support checksum for bootloaders. The following devices do support checksum operation for bootloaders:</p> <ul style="list-style-type: none"> <li>Zynq UltraScale+ MPSoC</li> <li>Versal adaptive SoC</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">copy</a> <address>	This attribute specifies that the image is to be copied to memory at specified address.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">core</a> <options>	<p>This attributes specifies which core executes the partition. The options for AMD Versal™ adaptive SoC are:</p> <ul style="list-style-type: none"> <li><code>a72-0</code></li> <li><code>a72-1</code></li> <li><code>r5-0</code></li> <li><code>r5-1</code></li> <li><code>psm</code></li> <li><code>aie</code></li> <li><code>r5-lockstep</code></li> </ul>	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">delay_handoff</a>	This attribute specifies that the hand-off to the subsystem/image is delayed.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">delay_load</a>	This attribute specifies that the loading of the subsystem/image is delayed.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">delay_auth</a>	Indicates that the authentication is done at a later stage. This helps bootgen to reserve space for hashes during partition encryption.	<ul style="list-style-type: none"> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>destination_device</code> <device_type>	This specifies if the partition is targeted for PS or PL. The options are: <ul style="list-style-type: none"> <li><code>ps</code>: the partition is targeted for PS (default).</li> <li><code>pl</code>: the partition is targeted for PL, for bitstreams.</li> </ul>	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<code>destination_cpu</code> <device_core>	Specifies the core on which the partition should be executed. <ul style="list-style-type: none"> <li><code>a53-0</code></li> <li><code>a53-1</code></li> <li><code>a53-2</code></li> <li><code>a53-3</code></li> <li><code>r5-0</code> (default)</li> <li><code>r5-1</code></li> <li><code>pmu</code></li> <li><code>r5-lockstep</code></li> </ul>	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<code>early_handoff</code>	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.	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<code>efuse_kek_iv</code> <filename>	Specifies the IV that is used to encrypt the corresponding key. <code>efuse_kek_iv</code> is valid with <code>keysrc=efuse_blk_key</code> .	<ul style="list-style-type: none"> <li>versal</li> </ul>
<code>efuse_user_kek0_iv</code> <filename>	Specifies the IV that is used to encrypt the corresponding key. <code>efuse_user_kek0_iv</code> is valid with <code>keysrc=efuse_user_blk_key0</code> .	<ul style="list-style-type: none"> <li>versal</li> </ul>
<code>efuse_user_kek1_iv</code> <filename>	Specifies the IV that is used to encrypt the corresponding key. <code>efuse_user_kek1_iv</code> is valid with <code>keysrc=efuse_user_blk_key1</code> .	<ul style="list-style-type: none"> <li>versal</li> </ul>
<code>encryption</code> <option>	Specifies the partition to be encrypted. Encryption algorithms are: zynq uses AES-CBC, while zynqmp and Versal use AES-GCM. The partition options are: <ul style="list-style-type: none"> <li><code>none</code>: Partition not encrypted.</li> <li><code>aes</code>: Partition encrypted using AES algorithm.</li> </ul>	<ul style="list-style-type: none"> <li>All</li> </ul>



Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>exception_level</code> <options>	Exception level for which the core should be configured. Options are: <ul style="list-style-type: none"> <li><code>e1-0</code></li> <li><code>e1-1</code></li> <li><code>e1-2</code></li> <li><code>e1-3</code></li> </ul>	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<code>familykey</code> <key file>	Specifies the family key.	<ul style="list-style-type: none"> <li>zynqmp</li> <li>fpga</li> </ul>
<code>file</code> <path/to/file>	This attribute specifies the file for creating the partition.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<code>fsbl_config</code> <options>	Specifies the sub-attributes used to configure the bootimage. Those sub-attributes are: <ul style="list-style-type: none"> <li><code>bh_auth_enable</code>: RSA authentication of the boot image is done excluding the verification of PPK hash and SPK ID.</li> <li><code>auth_only</code>: boot image is only RSA signed. FSBL should not be decrypted.</li> <li><code>opt_key</code>: Operational key is used for block-0 decryption. Secure Header has the opt key.</li> <li><code>pufhd_bh</code>: PUF helper data is stored in Boot Header (Default is <code>efuse</code>).</li> <li>PUF helper data file is passed to Bootgen using the <code>[puf_file]</code> option.</li> <li><code>puf4kmode</code>: PUF is tuned to use in 4k bit configuration (Default is 12k bit).</li> <li><code>shutter = &lt;value&gt;</code>: 32 bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time. Note that this shutter value must match the shutter value that was used during PUF registration.</li> </ul>	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<code>headersignature</code> <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.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">hivec</a>	<p>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.</p> <ul style="list-style-type: none"> <li><code>hivec</code>: exception vector table at 0xFFFF0000.</li> <li><code>lovec</code>: exception vector table at 0x00000000.</li> </ul>	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<a href="#">id</a> <id>	<p>This attribute specifies the following IDs based on the place its defined:</p> <ul style="list-style-type: none"> <li><code>pdi id</code> - within outermost/PDI parenthesis</li> <li><code>image id</code> - within image parenthesis</li> <li><code>partition id</code> - within partition parenthesis</li> </ul>	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">image</a>	Defines a subsystem/image.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">init</a> <filename>	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.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">keysrc</a>	<p>Specifies key source for encryption for Versal adaptive SoC. The keysrc can be specified for individual partitions.</p> <ul style="list-style-type: none"> <li><code>efuse_red_key</code></li> <li><code>efuse_blk_key</code></li> <li><code>bbram_red_key</code></li> <li><code>bbram_blk_key</code></li> <li><code>bh_blk_key</code></li> <li><code>user_key0</code></li> <li><code>user_key1</code></li> <li><code>user_key2</code></li> <li><code>user_key3</code></li> <li><code>user_key4</code></li> <li><code>user_key5</code></li> <li><code>user_key6</code></li> <li><code>user_key7</code></li> <li><code>efuse_user_key0</code></li> <li><code>efuse_user_blk_key0</code></li> <li><code>efuse_user_key1</code></li> <li><code>efuse_user_blk_key1</code></li> </ul>	<ul style="list-style-type: none"> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">keysrc_encryption</a>	Specifies the key source for encryption. The keys are: <ul style="list-style-type: none"> <li><code>efuse_gry_key</code>: Grey (Obfuscated) Key stored in eFUSE. See <a href="#">Gray/Obfuscated Keys</a></li> <li><code>bh_gry_key</code>: Grey (Obfuscated) Key stored in boot header.</li> <li><code>bh_blk_key</code>: Black Key stored in boot header. See <a href="#">Black/PUF Keys</a></li> <li><code>efuse_blk_key</code>: Black Key stored in eFUSE.</li> <li><code>kup_key</code>: User Key.</li> <li><code>efuse_red_key</code>: Red key stored in eFUSE. See <a href="#">Rolling Keys</a>.</li> <li><code>bbram_red_key</code>: Red key stored in BBRAM.</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> </ul>
<a href="#">load</a> <address>	Sets the desired load address for the partition in memory.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">metaheader</a>	This attribute is used to define encryption and authentication attributes for meta headers like keys, key sources, and so on.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">name</a> <name>	This attribute specifies the name of the image/subsystem.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">offset</a> <offset>	Sets the absolute offset of the partition in the boot image.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">parent_id</a>	This attribute specifies the ID for the parent PDI. This is used to identify the relationship between a partial PDI and its corresponding boot PDI.	<ul style="list-style-type: none"> <li>versal</li> </ul>
<a href="#">partition</a>	This attribute is used to define a partition. It is an optional attribute to make the BIF short and readable.	<ul style="list-style-type: none"> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">partition_owner</a> , <a href="#">owner</a> <option>	Owner of the partition which is responsible to load the partition. Options are: For Zynq/Zynq UltraScale+ MPSoC: <ul style="list-style-type: none"> <li><code>fsbl</code>: Partition is loaded by FSBL.</li> <li><code>uboot</code>: Partition is loaded by U-Boot.</li> </ul> For Versal: <ul style="list-style-type: none"> <li><code>plm</code>: partition loaded by PLM.</li> <li><code>non-plm</code>: partition is not loaded by PLM, but it is loaded by another entity like U-Boot.</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">pid</a> <ID>	Specifies the Partition ID. PID can be a 32-bit value (0 to 0xFFFFFFFF).	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<a href="#">pmufw_image</a> <image_name>	PMU firmware image to be loaded by BootROM, before loading the FSBL.	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<a href="#">ppkfile</a> <key filename>	Primary Public Key (PPK). Used to authenticate partitions in the boot image. See <a href="#">Using Authentication</a> for more information.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">presign</a> <sig_filename>	Partition signature (.sig) file.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>fpga</li> </ul>
<a href="#">pskfile</a> <key filename>	Primary Secret Key (PSK). Used to authenticate partitions in the boot image. See the <a href="#">Using Authentication</a> for more information.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">puf_file</a> <filename>	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.	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">reserve</a> <size in bytes>	Reserves the memory, which is padded after the partition.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">spk_select</a> <SPK_ID>	Specify an SPK ID in user eFUSE.	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>
<a href="#">spkfile</a> <filename>	Keys used to authenticate partitions in the boot image. See <a href="#">Using Authentication</a> for more information.	<ul style="list-style-type: none"> <li>All</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">spksignature</a> <signature_file>	Imports the SPK signature into an Authentication Certificate. See <a href="#">Using Authentication</a> . 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.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">split</a> <options>	<p>Splits the image into parts, based on the mode. Split options are:</p> <ul style="list-style-type: none"> <li>Slaveboot: Supported for Zynq UltraScale+ MPSoC only. Splits as follows: <ul style="list-style-type: none"> <li>Boot Header + Bootloader</li> <li>Image and Partition Headers</li> <li>Rest of the partitions</li> </ul> </li> <li>normal: Supported for zynq, zynqmp, and versal. Splits as follows: <ul style="list-style-type: none"> <li>Bootholder + Image Headers + Partition Headers + Bootloader</li> <li>Partition1</li> <li>Partition2 and so on</li> </ul> </li> </ul> <p>Along with the split mode, output format can also be specified as <code>bin</code> or <code>mcs</code>.</p> <p><b>Note:</b> The option split mode normal is same as the command line option split. This command line option is deprecated. Split ulaveboot is supported only for Zynq UltraScale+ MPSoC.</p>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">sskfile</a> <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.	<ul style="list-style-type: none"> <li>All</li> </ul>
<a href="#">startup</a> <address>	Sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">trustzone</a> <option>	<p>The trustzone options are:</p> <ul style="list-style-type: none"> <li><code>secure</code></li> <li><code>nonsecure</code></li> </ul>	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>

Table 32: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<a href="#">type</a> <options>	This attribute specifies the type of partition. The options are: <ul style="list-style-type: none"> <li>• <code>bootloader</code></li> <li>• <code>pmcdata</code></li> <li>• <code>cdo</code></li> <li>• <code>cfi</code></li> <li>• <code>cfi-gsc</code></li> <li>• <code>bootimage</code></li> </ul>	<ul style="list-style-type: none"> <li>• versal</li> </ul>
<a href="#">udf_bh</a> <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.	<ul style="list-style-type: none"> <li>• zynq</li> <li>• zynqmp</li> </ul>
<a href="#">udf_data</a> <data_file>	Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate.	<ul style="list-style-type: none"> <li>• zynq</li> <li>• zynqmp</li> </ul>
<a href="#">userkeys</a> <filename>	The path to the user keyfile.	<ul style="list-style-type: none"> <li>• versal</li> </ul>
<a href="#">xip_mode</a>	Indicates eXecute In Place (XIP) for FSBL to be executed directly from QSPI flash.	<ul style="list-style-type: none"> <li>• zynq</li> <li>• zynqmp</li> </ul>

# Using Bootgen GUI

Bootgen has both a GUI and a command-line option. The GUI option is available in the AMD Vitis™ IDE as a wizard. The functionality in the Vitis IDE 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.

---

## Launch Bootgen GUI

To create a boot image using the IDE, do either of the following:

- Select the application project in the **Project Navigator** or **C/C++ Projects** view and right-click **Create Boot Image**.
- Alternatively, click **Vitis** → **Create Boot Image** → **Target Device Family**.

The supported device families are

- AMD Zynq™ and AMD Zynq™ AMD UltraScale+™
- AMD Versal™

---

## Bootgen GUI for Zynq 7000 and Zynq UltraScale+ Devices

After you launch Bootgen GUI for AMD Zynq™ and Zynq AMD UltraScale+™, the Create Boot Image dialog box opens, with default values pre-selected from the context of the selected project.

Figure 7: Create Boot Image for Zynq and Zynq UltraScale+ Devices

**Create Boot Image**  
Creates Zynq MP Boot Image in .bin format from given FSBL elf and partition files in specified output folder.

Architecture: **Zynq MP**

☐ Create new BIF file ☒ Import from existing BIF file

Import BIF file path:  **Browse...**

**Basic** **Security**

Output BIF file path:  **Browse...**

UDF data:  **Browse...**

☐ Split **Output format:** **BIN**

Output path:  **Browse...**

**Boot image partitions**

File path	Encrypted	Authenticated
(bootloader) /tmp/vitis_project/zcu102/export/zcu102	none	none
/tmp/vitis_project/zynqmp/Debug/zynqmp.elf	none	none

**Add**  
**Delete**  
**Edit**  
**Up**  
**Down**

**Preview BIF Changes** **Cancel** **Create Image**

- 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.
1. 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**.
2. In the Boot image partitions, click the **Add** button to add additional partition images.
3. Create offset, alignment, and allocation values for partitions in the boot image, if applicable.

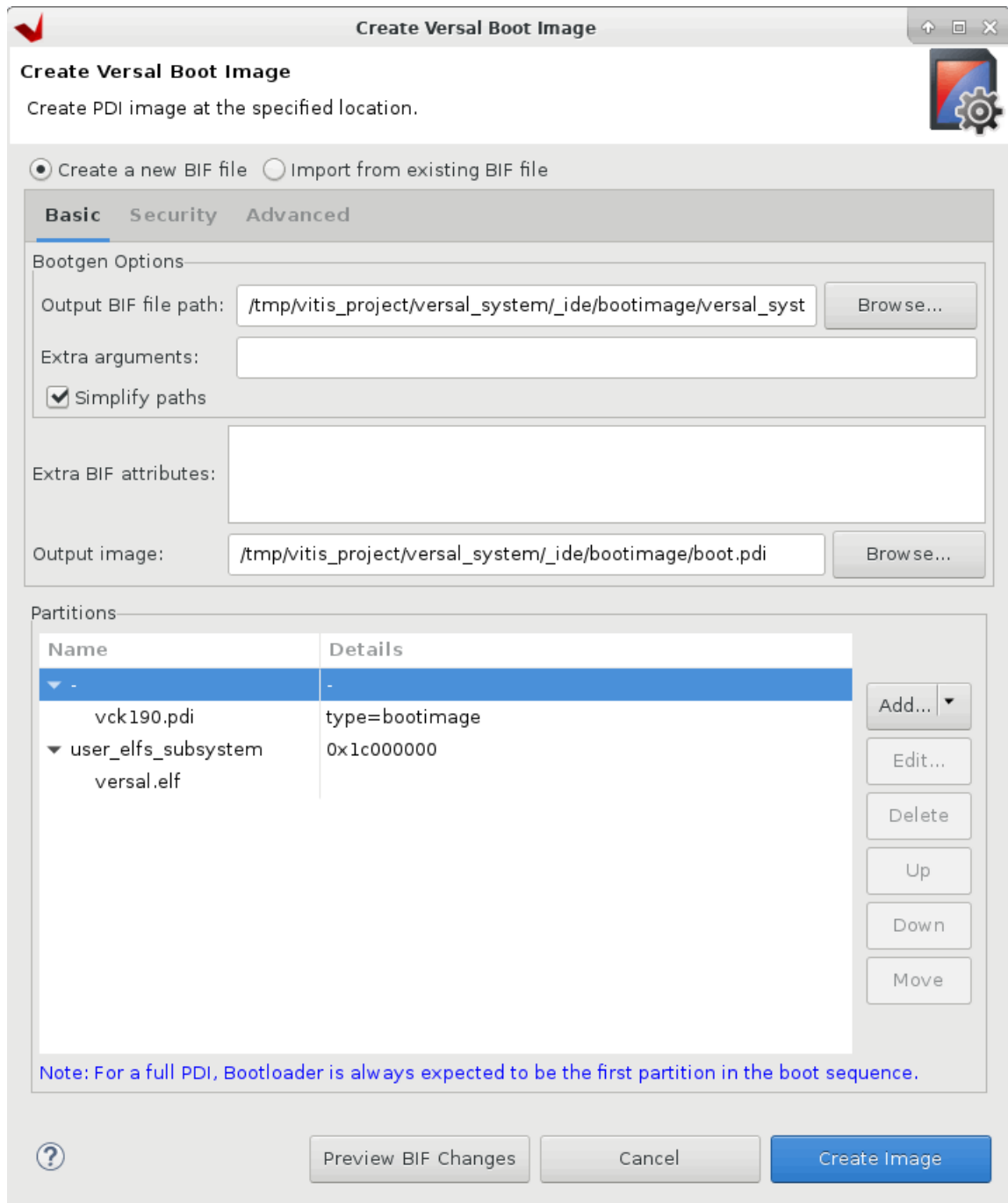
The output file path is set to the `/bif` folder under the selected application project by default.
4. From the Security tab, you can specify the attributes to create a secure image. This security can be applied to individual partitions as required.
  - a. To enable authentication for a partition, check the **Use Authentication** option, then specify the PPK, SPK, PSK, and SSK values. See the [Using Authentication](#) topic for more information.
  - b. To enable encryption for a partition, select the Encryption view, and check the **Use Encryption** option. See [Using Encryption](#) for more information.
5. 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 GUI Options for Versal Adaptive SoCs

After the user launches Bootgen GUI for AMD Versal™, the Create Boot Image dialog box opens, with default values pre-selected from the context of the selected project.

Figure 8: Bootgen GUI for Versal Adaptive SoCs



1. Populate the Create Boot Image dialog box with the following information:
  - a. Select either **Create a BIF file** or **Import an existing BIF file**  
**Note:** AMD Vivado™ generated BIF can be found at <design>.runs/impl\_1/ directory.
  - b. From the Basic tab, specify the **Output BIF file path**.

- c. Specify the **Output Image**.
2. In the Boot image partitions, click the **Add** button to add additional partition images.
3. 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.
4. From the **Security** tab, you can specify the overall attributes to create a secure image. Partition security attributes can be updated when you select the **partition** and click **Edit** button, then go to **Security** tab.  
See [Using Authentication](#) and [Using Encryption](#) for more information.
5. 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**.
6. Contents in Extra Bif attributes dialogue and the **Edit partition → Extra Partition attributes** is appended to the overall BIF file or the partition. You can use these fields to add custom attributes if they are not supported by the Bootgen GUI.

---

## 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 Vitis IDE. In the standard install of the Vitis software platform, the XSCT 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 standalone 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.

---

## 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 AMD device uses that command:

- `zynq`: AMD Zynq™ 7000 SoC device
- `zynqmp`: AMD Zynq™ UltraScale+™ MPSoC device

- `fpga`: Any 7 series and above devices
- `versal`: Versal adaptive SoC

For more information, see [Chapter 10: Command Reference](#).

**Table 33: Bootgen Command and Descriptions**

Commands	Description and Options	Used by
<code>arch &lt;type&gt;</code>	AMD device architecture. Options: <ul style="list-style-type: none"> <li>• <code>zynq</code> (default)</li> <li>• <code>zynqmp</code></li> <li>• <code>fpga</code></li> <li>• <code>versal</code></li> </ul>	<ul style="list-style-type: none"> <li>• All</li> </ul>
<code>authenticatedjtag &lt;options&gt;</code>	Used to enable JTAG during secure boot. The arguments are: <ul style="list-style-type: none"> <li>• <code>rsa</code></li> <li>• <code>ecdsa</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>versal</code></li> </ul>
<code>bif_help</code>	Prints out the BIF help summary.	<ul style="list-style-type: none"> <li>• All</li> </ul>
<code>dual_qspi_mode &lt;configuration&gt;</code>	Generates two output files for dual QSPI configurations: <ul style="list-style-type: none"> <li>• <code>parallel</code></li> <li>• <code>stacked &lt;size&gt;</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>zynq</code></li> <li>• <code>zynqmp</code></li> <li>• <code>versal</code></li> </ul>
<code>dual_ospi_mode stacked &lt;size&gt;</code>	Generates two output files for stacked configuration.	<ul style="list-style-type: none"> <li>• <code>versal</code></li> </ul>
<code>dump &lt;options&gt;</code>	Dumps the partition or boot header as per options specified. <ul style="list-style-type: none"> <li>• <code>empty</code>: Dumps the partitions as binary files.</li> <li>• <code>bh</code>: Dumps boot header as a binary file.</li> <li>• <code>plm</code>: Dumps PLM as a binary file.</li> <li>• <code>pmc_cdo</code>: Dumps PMC CDO as a binary file.</li> <li>• <code>boot_files</code>: Dumps boot header, PLM and PMC CDO as three separate binary files.</li> <li>• <code>slave_pdis</code>: Dumps Slave PDIs for SSI technology use cases</li> </ul> <p>The slave PDIs are not dumped by default. This option should be used if you want to debug or analyze the slave PDI separately.</p>	<ul style="list-style-type: none"> <li>• <code>versal</code></li> </ul>
<code>dump_dir</code>	Dumps components in specified directory.	<ul style="list-style-type: none"> <li>• <code>versal</code></li> </ul>
<code>efuseppkbits &lt;PPK_filename&gt;</code>	Generates a PPK hash for eFUSE.	<ul style="list-style-type: none"> <li>• <code>zynq</code></li> <li>• <code>zynqmp</code></li> <li>• <code>versal</code></li> </ul>
<code>encrypt &lt;options&gt;</code>	AES Key storage in device. Options are: <ul style="list-style-type: none"> <li>• <code>bbram</code> (default)</li> <li>• <code>efuse</code></li> </ul>	<ul style="list-style-type: none"> <li>• <code>zynq</code></li> <li>• <code>fpga</code></li> </ul>

Table 33: Bootgen Command and Descriptions (cont'd)

Commands	Description and Options	Used by
<a href="#">encryption_dump</a>	Generates encryption log file, <code>aes_log.txt</code> .	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">fill</a> <hex_byte>	Specifies the fill byte to use for padding.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">generate_hashes</a>	Generates file containing padded hash: <ul style="list-style-type: none"> <li>Zynq devices: SHA-2 with PKCS#1v1.5 padding scheme</li> <li>Zynq UltraScale+ MPSoC: SHA-3 with PKCS#1v1.5 padding scheme</li> <li>Versal adaptive SoC: SHA-3 with PSS padding scheme</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">generate_keys</a> <key_type>	Generate the authentication keys. Options are: <ul style="list-style-type: none"> <li><code>pem</code></li> <li><code>rsa</code></li> <li><code>obfuscatedkey</code></li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">h, help</a>	Prints out help summary.	<ul style="list-style-type: none"> <li>All</li> </ul>
<a href="#">image</a> <filename(.bif)>	Provides a boot image format ( <code>.bif</code> ) file name.	<ul style="list-style-type: none"> <li>All</li> </ul>
<a href="#">log</a> <level_type>	Generates a log file at the current working directory with following message types: <ul style="list-style-type: none"> <li><code>error</code></li> <li><code>warning</code> (default)</li> <li><code>info</code></li> <li><code>debug</code></li> <li><code>trace</code></li> </ul>	<ul style="list-style-type: none"> <li>All</li> </ul>
<a href="#">nonbooting</a>	Create an intermediate boot image.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<a href="#">o</a> <filename>	Specifies the output file. The format of the file is determined by the file name extension. Valid extensions are: <ul style="list-style-type: none"> <li><code>.bin</code> (default)</li> <li><code>.mcs</code></li> <li><code>.pdi</code></li> </ul>	<ul style="list-style-type: none"> <li>All</li> </ul>
<a href="#">overlay_cdo</a> <filename>	CDO overlay option provides a way to modify CDO files after they are generated.	versal
<a href="#">p</a> <partname>	Specify the part name used in generating the encryption key.	<ul style="list-style-type: none"> <li>All</li> </ul>

Table 33: Bootgen Command and Descriptions (cont'd)

Commands	Description and Options	Used by
<code>padimageheader</code> <option>	Pads the image headers to force alignment of following partitions. Options are: <ul style="list-style-type: none"> <li>0</li> <li>1 (default)</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> </ul>
<code>process_bitstream</code> <option>	Specifies that the bitstream is processed and outputs as <code>.bin</code> or <code>.mcs</code> . <ul style="list-style-type: none"> <li>For example, if encryption is selected for bitstream in BIF file, the output is an encrypted bitstream.</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> </ul>
<code>read</code> <options>	Used to read boot headers, image headers, and partition headers based on the options. <ul style="list-style-type: none"> <li><code>bh</code>: To read boot header from bootimage in human readable form</li> <li><code>ih</code>: To read image header table from bootimage</li> <li><code>ih</code>: To read image headers from bootimage.</li> <li><code>pht</code>: To read partition headers from bootimage</li> <li><code>ac</code>: To read authentication certificates from bootimage</li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<code>split</code> <options>	Splits the boot image into partitions and outputs the files as <code>.bin</code> or <code>.mcs</code> . <ul style="list-style-type: none"> <li>Bootheder + Image Headers + Partition Headers + <code>Fsbl.elf</code></li> <li><code>Partition1.bit</code></li> <li><code>Partition2.elf</code></li> </ul>	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<code>spksignature</code> <filename>	Generates an SPK signature file.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<code>verify</code> <filename>	This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image is verified against the available partitions.	<ul style="list-style-type: none"> <li>zynq</li> <li>zynqmp</li> <li>versal</li> </ul>
<code>verify_kdf</code>	This option is used to validate the Counter Mode KDF used in Bootgen for generation AES keys.	<ul style="list-style-type: none"> <li>zynqmp</li> <li>versal</li> </ul>
<code>w</code> <option>	Specifies whether to overwrite the output files: <ul style="list-style-type: none"> <li><code>on</code> (default)</li> <li><code>off</code></li> </ul> <p><b>Note:</b> The <code>-w</code> without an option is interpreted as <code>-w on</code>.</p>	<ul style="list-style-type: none"> <li>All</li> </ul>
<code>zynqmpes1</code>	Generates a boot image for ES1 (1.0). The default padding scheme is ES2 (2.0).	<ul style="list-style-type: none"> <li>zynqmp</li> </ul>

# Boot Time Security

AMD supports secure booting on all devices using latest authentication methods to prevent unauthorized or modified code from being run on AMD devices. AMD 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 AMD 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, your 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), that 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 an AMD 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.

### Versal Adaptive SoC Security

On AMD Versal™ adaptive SoCs, secure boot ensures the confidentiality, integrity, and authentication of the firmware and software loaded onto the device. The root of trust starts with the PMC ROM, that authenticates and/or decrypts the PLM software. Now that the PLM software is trusted, the PLM handles loading the rest of the firmware and software in a secure manner. Additionally, if secure boot is not desired then software can at least be validated with a simple checksum.

See *Versal Adaptive SoC Technical Reference Manual* (AM011) for more information. Also, see the *Versal Adaptive SoC Security Manual* (UG1508). This manual requires an active NDA to be downloaded from the Design Security Lounge.

---

## Using Encryption

Secure booting, that 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, AMD 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:

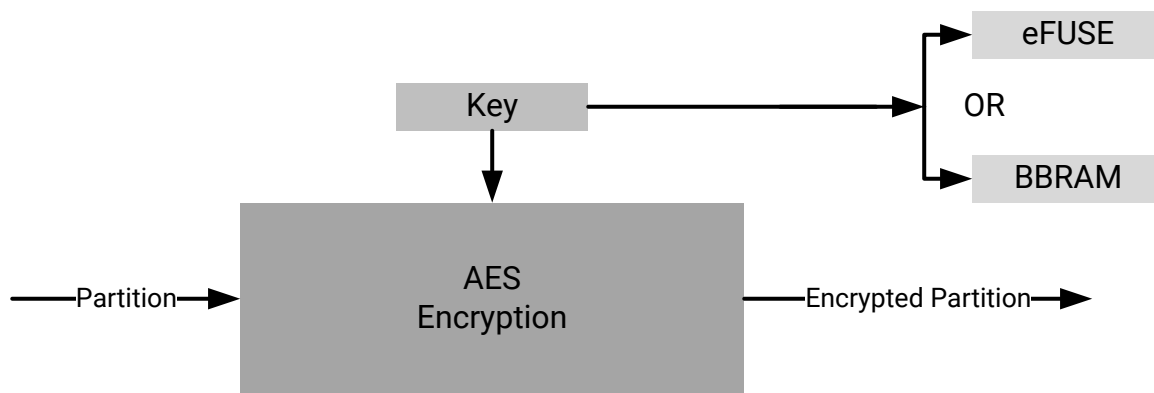
- Works by repeating the same defined steps multiple times
- Uses a secret key encryption algorithm
- Operates on a fixed number of bytes



## 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.

Figure 9: Encryption Process Diagram

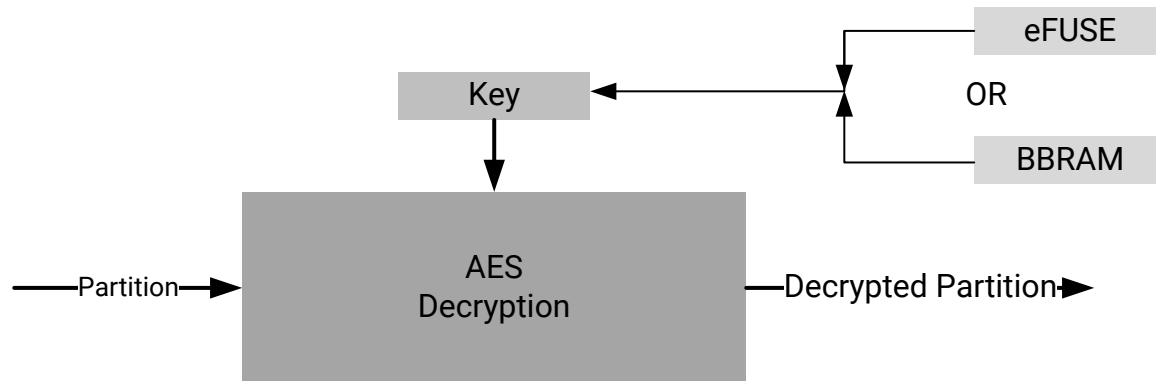


X21274-062320

## 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 starts 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.

Figure 10: Decryption Process Diagram



X21274-062320

## Encrypting Zynq-7000 Device Partitions

AMD Zynq™ 7000 SoC devices use the embedded, Programmable 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 11: Sample Key File

```
Device      xc7z020clg484;
Key 0       f878b838d8589818e868a828c8488808
Key StartCBC 5C9D95ECBFEC8A1F12A8EB312362C596
Key HMAC    00001111222233334444555566667777
```

## Encrypting Zynq MPSoC Device Partitions

The AMD 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 (`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 `opt_key` is used to specify operational key usage. See [fsbl\\_config](#) for more information about the `opt_key` value that is an argument to the `fsbl_config` attribute. The following is an example of using the `opt_key` attribute.

```
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 `Key Opt` as shown in the following example.

Figure 12: Operational Key

```
Device      xczu9eg;
Key 0       9C42D9B74B633132F57C381D5CA4C7DF0829382CDBC455CDA08ECA62EB11D19D;
IV 0        42D3818AC135A365EDBD5316;
Key Opt     36AD8321ECA72E9F88E4F3A85ACD9ACDA27D1F50773E24B95067BA3BA75A3A62;
```

Bootgen generates the encryption key file. The operational key `opt_key` is then generated in the .nky file, if `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.

```
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 (does not read) 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.
  - No Key/IV pair should be repeated in any of the aes key files given in a single bif except the Key0 and IV0.

## 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 AMD 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 [Chapter 8: 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 the first partition. For example, in the case of a full boot image, the first partition is the bootloader.
- 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.

Figure 13: Sample Key File

```
Device      xczu9eg;
Key 0      AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
IV 0      11198912D243EF0AFEAC8970;
Key 1      C023E238AC903111DEF0AABB98C1CCDDEEFF021001289011198C1E238AC34012;
IV 1      111DEF0AABBCCDDEEFF00112;
Key 2      11456A9B8764DE111444C023E238A98C1CCC9031177112E01289011198CFF010;
IV 2      9C64778CBAF48D6DDE13749B;
Key Opt    229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
```

## 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 AMD Security Group. For more information, see [familykey](#). To generate an obfuscated key, Bootgen takes the following inputs from the BIF file.

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

The command to generate the obfuscated key is:

```
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.

```
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 [Chapter 8: 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), IV0, 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, and so on in the same directory of the key file meant for that partition.

The following snippet shows a sample encryption key file:

```
all:
{
    [keysrc_encryption] bbram_red_key
    // FSBL (Partition-0)
    [
        bootloader,
        destination_cpu = a53-0,
        encryption = aes,
        aeskeyfile = key_p0.nky

    ]fsbla53.elf

    // application (Partition-1)
    [
        destination_cpu = a53-0,
        encryption = aes,
        aeskeyfile = key_p1.nky

    ]hello.elf
}
```

- The partition `fsbla53.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`.

## Encrypting Versal Device Partitions

The AMD Versal™ device uses the AES-GCM core, which has support for a 256-bit key. When creating a secure image, each partition in a boot image can be optionally encrypted. Key source and aes key file are the prerequisites for encryption.

**Note:** For Versal adaptive SoC, it is mandatory to specify AES key file and the key source for each partition when encryption is enabled. Based on the key source used, same Key0 should be used in the aes key files specified respectively and vice-versa.

### Key Management

Good key management practice includes minimizing the use of secret or private keys. This can be accomplished by using different key/IV pairs across different partitions in the boot image. 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.

```
all: {
    image
    {
        {type=bootloader, encryption=aes, keysrc=bbram_red_key,
        aeskeyfile=plm.nky, dpacm_enable, file=plm.elf}
        {type=pmcdata, load=0xf2000000, aeskeyfile = pmc_data.nky,
```



```

file=pmc_data.cdo}
{core=psm, file=psm.elf}
{type=cdo, encryption=aes, keysrc=bbram_red_key,
aeskeyfile=ps_data.nky, file=ps_data.cdo}
{type=cdo, file=subsystem.cdo}
{core=a72-0, exception_level = el-3, file=a72-app.elf}
}
}

```

## ***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. You can generate the boot images with rolling keys using Bootgen. The BIF attribute blocks is used to specify the pattern to create multiple smaller blocks for encryption.

**Note:** For Versal adaptive SoC, a default key rolling is done on 32 KB of data. The key rolling you choose with the attribute blocks is applied in each 32 KB chunk. This is to compliment the hashing scheme used. If the DPA key rolling countermeasure is enabled, boot time is impacted. Refer to the boot time estimator spreadsheet for calculations.

```

all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    metaheader
    {
        encryption = aes,
        keysrc = bbram_red_key,
        aeskeyfile = efuse_red_metaheader_key.nky,
        dpacm_enable
    }

    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition
        {
            id = 0x01, type = bootloader,
            encryption = aes,
            keysrc = bbram_red_key,
            aeskeyfile = bbram_red_key.nky,
            dpacm_enable,
            blocks = 4096(2);1024;2048(2);4096(*),
            file = plm.elf
        }
        partition
        {
            id = 0x09, type = pmcdata, load = 0xf2000000,
            aeskeyfile = pmcdata.nky,
            file = pmc_data.cdo
        }
    }
}

```

```

image
{
    name = lpd, id = 0x4210002
    partition
    {
        id = 0x0C, type = cdo,
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = key1.nky,
        dpacm_enable,
        blocks = 8192(20);4096(*),
        file = lpd_data.cdo
    }
    partition
    {
        id = 0x0B, core = psm,
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = key2.nky,
        dpacm_enable,
        blocks = 4096(2);1024;2048(2);4096(*),
        file = psm_fw.elf
    }
}

image
{
    name = fpd, id = 0x420c003
    partition
    {
        id = 0x08, type = cdo,
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = key5.nky,
        dpacm_enable,
        blocks = 8192(20);4096(*),
        file = fpd_data.cdo
    }
}
}

```

**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.

**Key Generation**

Bootgen can generate 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 you want 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. The following conditions apply.

- 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 the 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.
- If no key file is specified and encryption is opted for a partition, Bootgen by default generated an `aes` key file with the name of the partition. By doing this, Bootgen makes sure that a different `aeskeyfile` is used for each partition.
- Bootgen enables the usage of unique key files for each of the partition created due to multiple loadable sections by reading/generating key file names appended with ".1", ".2"... "n" and so on in the same directory of the key file meant for that partition.

## 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. Example:

```
test:
{
  bh_kek_iv = black_iv.txt
  bh_keyfile = black_key.txt
  puf_file = pufdata.txt
  boot_config {puf4kmode}
  image
  {
    {type=bootloader, encryption = aes, keysrc=bh_blk_key, pufhd_bh,
aeskeyfile = red_grey.nky, file=plm.elf}
    {type=pmcdata,load=0xf2000000, aeskeyfile = pmcdata.nky,
file=pmc_data.cdo}
    {core=psm, file=psm.elf}
    {type=cdo, file=ps_data.cdo}
    {type=cdo, file=subsystem.cdo}
    {core=a72-0, exception_level = el-3, file=hello_world.elf}
  }
}
```

## Meta Header Encryption

For a Versal adaptive SoC, Bootgen encrypts the meta header when encryption is specifically mentioned under the "metaheader" attribute. The `aeskeyfile` to be used can be specified in the `bif` using the parameters under "metaheader." A snippet of the usage is shown below.

**Note:** Meta Header encryption includes all the headers except the Image Header Table.

```
metaheader
{
  encryption = aes,
  keysrc = bbam_red_key,
  aeskeyfile = headerkey.nky,
}
```

The following conditions apply.

- If a specific `aeskeyfile` is not specified for the meta header, Bootgen generates a file named `meta_header.nky`, and uses it during encryption.
- If a boot loader is present in the `bif`, it is mandatory to encrypt boot loader to encrypt meta header. For a partial PDI, meta header can be optionally chosen to be encrypted.
- To ensure the correctness of Image Header Table, it is added as additional authenticated data when encrypting the meta header.

---

## 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 the internal key space). There is an alternative form of authentication in the form of Rivest-Shamir-Adleman (RSA). 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. AMD 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 AMD 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:

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

For device-specific Authentication information, see the following:

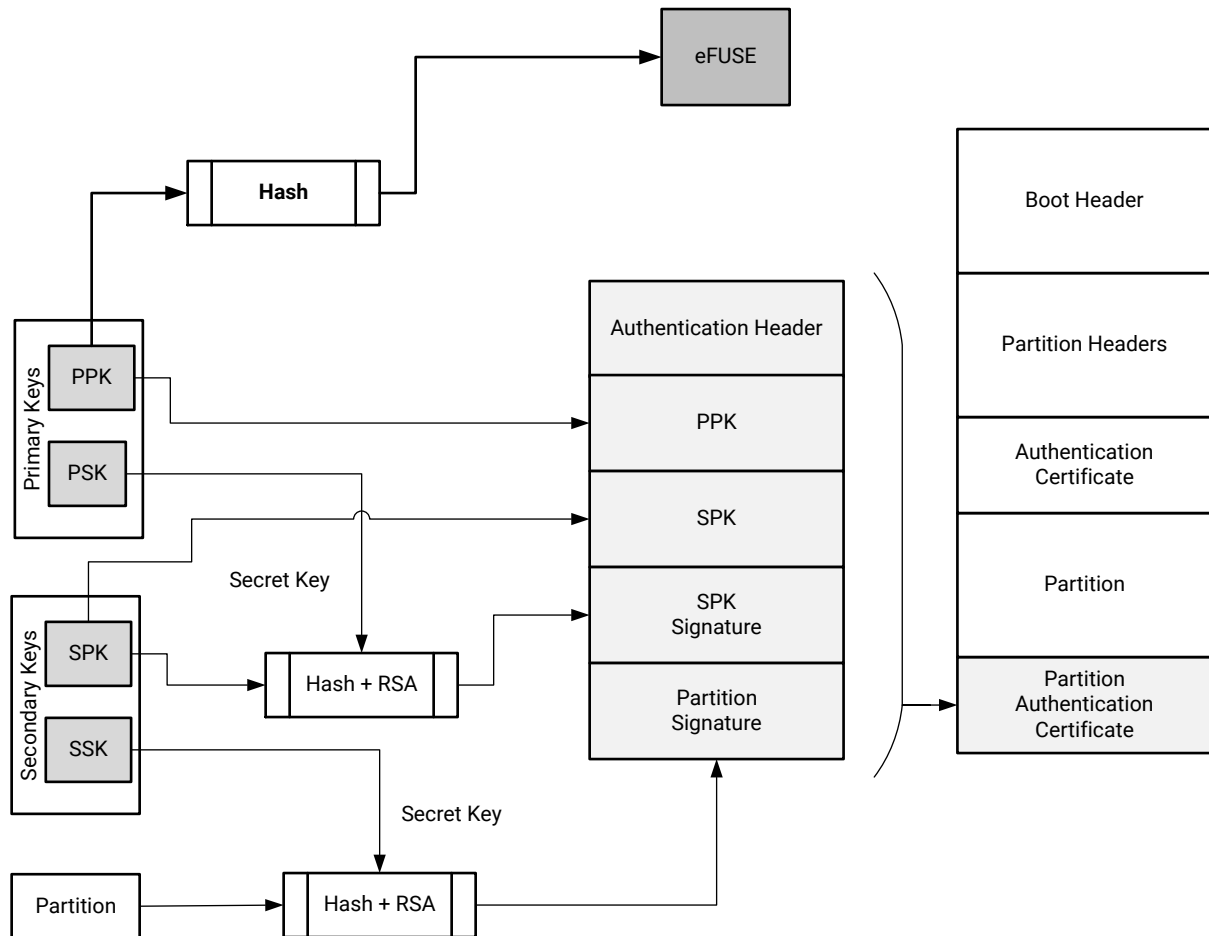
- [Zynq-7000 Authentication Certificates](#)
- [Zynq UltraScale+ MPSoC Authentication Certificates](#)
- [Versal Adaptive SoC Authentication Certificates](#)

## 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 or prepended to each partition that is opted for authentication depending on the device.
5. PPK is hashed and stored in eFUSE.

Figure 14: RSA Partition Signature



X21278-080618

The following table shows the options for Authentication.

Table 34: Supported File Formats for Authentication Keys

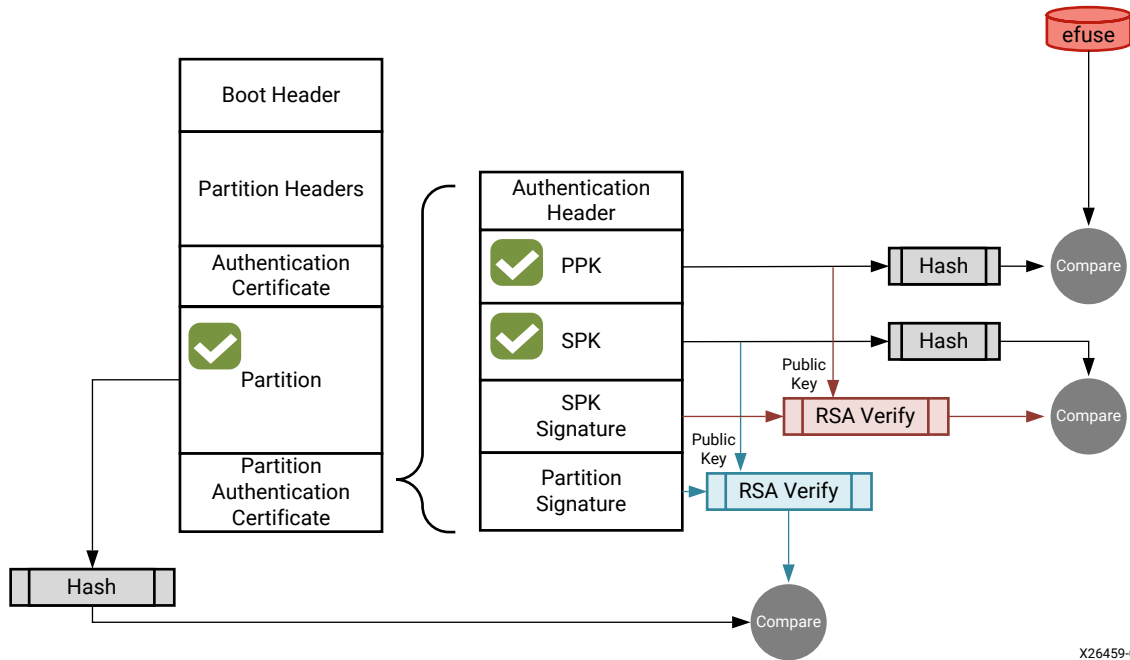
Key	Name	Description	Supported File Format
PPK	Primary Public Key	This key is used to authenticate a partition. It should always be specified when authenticating a partition.	*.pem *.pub
PSK	Primary Secret Key	This key is used to authenticate a partition. It should always be specified when authenticating a partition.	*.pem
SPK	Secondary Public Key	This key, when specified, is used to authenticate a partition.	*.pem *.pub
SSK	Secondary Secret Key	This key, when specified, is used to authenticate a partition.	*.pem pub

## 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 PPK
  - 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 SPK.
  - d. Compare the hashes from step (b) and step (c)
  - e. If same, then partition is trusted, else secure boot fail

Figure 15: Verification Flow Diagram



Bootgen can create a 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 AMD 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 35: Authentication Signatures

Which Authentication Certificate (AC)?	Signature	SHA Algorithm and SPK eFUSE	Secret Key used for Signature Generation
Partitions header AC (loaded by FSBL/FW)	SPK Signature	If SPKID eFUSEs, then Keccak; If User eFUSE, then NIST	PSK
	BH Signature	Always Keccak	SSK <sub>header</sub>
	Header Signature	Always Nist	SSK <sub>header</sub>
BootLoader (FSBL) AC (loaded by ROM)	SPK Signature	Always Keccak; Always SPKID eFUSE for SPK	PSK
	BH Signature	Always Keccak	SSK <sub>Bootloader</sub>
	FSBL Signature	Always Keccak	SSK <sub>Bootloader</sub>
Other Partition AC (loaded by FSBL FW)	SPK Signature	If SPKID eFUSEs then Keccak; If User eFUSE then NIST	PSK
	BH Signature	Always Keccak padding	SSK <sub>Partition</sub>
	Partition Signature	Always NIST padding	SSK <sub>Partition</sub>

## Examples

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

```
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_p1.pem
    ] fsbla53.elf

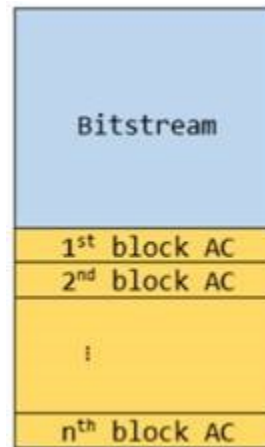
    // ATF (Partition-1)
    [
        destination_cpu = a53-0,
        authentication = rsa,
        exception_level = el-3,
        trustzone = secure,
        spk_id = 0x02,
        sskfile = secondary_p2.pem
    ] bl31.elf

    // UBOOT (Partition-2)
    [
        destination_cpu = a53-0,
        authentication = rsa,
        exception_level = el-2,
        spk_id = 0x03,
        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.

Figure 16: Bitstream Authentication Using External Memory



## 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.

```
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 = 0x100,
        sskfile = ssk3.pem
    ] application.elf
}
```

```
[
    destination_cpu = a53-0,
    authentication = rsa,
    spk_select = user-efuse,
    spk_id = 0x8,
    sskfile = ssk4.pem
] application2.elf
}
```

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

Partitions loaded by CSU ROM always uses `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 it is 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.

**Note:** In the above example, FSBL and application2 use the same `spk_id`. But these two keys can be revoked separately, because one is checked against the SPK\_ID eFUSE and the other is checked against User eFUSE.

## 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.

Figure 17: Sample RSA Private Key File

```

-----BEGIN RSA PRIVATE KEY-----
MIIEKAAIBAAKCAgEA4ppimme6TvPT5+JB2CgXQLU9AyStbnEr2lEJu+ZpR9H25Plq
6KbOcFuV6q3EKvI5PJsMS0yHpVr/1l/uTPxyUT6Im5goMyaskz0PS3xTWuYoSDba
YD50ZlPi5xBrswWvys6YcIbLTbk2+o86o0Rr/sdQtLR0pbsLfuBFoKMEsKl9N12k
E1l6DMlTjh9KspZozmj7yew2Rm857QqOp8sulVi4qdtIr58+MoQxeTEHcN+zuq4
drlUsUqX3msVb9z0rRwYrBVtSksWr5d+xj+cAUpiPjeMGRXg00L6gEGGPTjncQtG
YFCoCFCBL4JknHF/yMyV7f6wh2xtkKbme+Kuovcz/pQVKEGELkQ9kjweBf5c8Vmk
b13NvkrAUOXYLM+pyOuY/PGjtz6B5W964LOcrT+TRROi4FGotYzk2XmJtODO5dYH
Lw58IOT3zAYwaC/98bUDGYP6kJ9+YqprerLmZU55Ew30PPodjHYihLmBjlpvmu4g
oZ9tXJPch/urk/tv3e53P2JhWKwdb72FUi8hEgSkCWWAFfJwCVFWaTettzGlhtz+
Ww3eBAQI9fFbgr6YERwxOOLoparQiZPaC/8XG8u0bTE3Mdv5JK/IIOAqVnTl7Dfs
QKzTZap8+Iwx/vuaWAlD0qYCDKKKmlGGz5bQhEgRnk0I/lpOKIlPRL8wH0CAWEA
AQKCAgA3qhscuOxgZq8gYEkyCy67G4pgUks0PSK7n3qXqNMI7FvtToO/opJHUYgz
PPpaxMRHCgNsH+GWchM08gDU8pKWeJkQNS8FWr0jPZolyTpkFVDiC/M6KI+IuEZ9E
iZkbQgNb+4Iq6kvYz02/gR2za6Rn0shli3q4F4mMYkVYX5NQXmI/D0a2phiANDQX
r0IObnvY0SvppHynXIKU7UTMutPRlsdhpuFYMXjnOuWErzJbPOimrAzoF03FA7Y
eU+ryghk2ekJpL3TKTzqZ3mh85A8FOyrQfPtWZl/6A0nInF6apclxHpGQKn2WoEV
DZ/vekYcqn0OGKl+qtkDVqx5tEaXlXGlc0PBWg5aofkpNZ0K0wOG61CueNvATcJ9
RoMq7c7zZOYh4S2WgSjP3a8neGcnhG0T6BGYCGjPXRW2Y6ri/7lRdCOBVSc3zS8p
IVKAbp13PIgZlhMnxdC60RPh8dhXRR0Tua3+1SyGx37Ad9260UeHHJlpz28DkzTg
CY7RU5SDSh6wDuDbhelu4nzZDGWeKq9zeAzXGZhIn0zcxpWvG54uHTHnNgBEFJ2S
ZS78sq4aYiZCiW/PrqKgg8wBygKcEtr3/LcAm4r3pl9mHk1555QQNDpk+ba+3GLp
bEy0889KwCyPKfWY5pl6VnGLycxe/TofMDCHQARA2wLRn1sQQKCAQEAS0nn83su
OYN9oc22owfm/MHGJ6mFi5LpRtGylWbcAbDsZs7rjQ4IZ46JQlMpiQ10IpnBvub7
sW0FUX7sVo0XZMSl+EpsZDq021+7hY6+MGALtPpg9n2Jz9lfcYVXfNqv5SiMv6Te
6/jur69KiwhztYfi7JK4GGUdcCWYAwMTdgm3pQDDH99Vp436klvk4lMyjeQaIpO/
FzkiklFYN84j9jvtagoMk0fzaickiOGSS4ciOdS3DEgGC9x1hDkIs9UFPk1Pfw+7
qYnsT7XiwoTCBrvQl1lKp5fLZUhsRrsIQV82u44IPfcU3XWgcyInSGx0RfSV5RWov
v9sJFVvFlXE5EQKCAQEAS7nFNK5gbPKA0nxKTeM1ZMHP9/YqRxpj5irmXmrF54cn
s2PpG/dvbJBXILAd9hsSYjW8FNY5ehJhL9IQzEVavFr8SAvu2FyI9MN0d9wUvpJG
55Xx9K090uSzaXZVimV/5xumbnynwx22wgxslSAYoNy+8soviZlXQXzZeUaohaM
VVuLlHdRzE0afrcFsnfugIDl72MbI4t2cRTftek/iYAvF9bkO76upkPmWu4V7yFT
of9QFkq8qBRthEpvaKNTObpU5TrzssUH3rYXVnA2gpEXEJdeVVFYzSLf4SC45mxe
Gpp37pYetPBKvUesUevQ70IeoiCGRXFgC9TPmYwRQKCAQEASD1CoPbAD+7ejVsD
a4fFX2K+7rin86A4v1q9hlzAK8n6jhyzeRpgIh7jgFiaj9hRnppVx3pdSD+6DJGh
UTV+a+fcuMnBVGqK/3+ZYhvfK2z/rqJyUuzFXDxWYROANz7GY5seKDC2fhgEG0dV
DIg6XV5sGvsuQJyj+HE0x0sDPlCxe9fyNrWEGvQkzXgX64gXlmvXZPbs//F3EIne
680lkyz3d1LEJ2wJ3V2pdc0BnvE4l75Kl/f9zCTgDtK6m7/Q0quOMreyJf/HWyy
UmLP0BdlAogfdIkApR0rKvym7milGQUMWXAq8sTS1FpPXYI4TpFwiZaXAg2a9w3
qdKVsQKCAQBH8nolofT/mxulsBY9ikDSrvPB0U6qe8UPC3zNmowyy25nv8jd/opp
iLgxjdLMkuieJ7ajluwq8GbQ5iLZcEfrs8yR9L/SG0HcES0QjKDZzAuHDoIVNuAS
CoS2dse4nv26zjn10s2BvmHvvuI3/BVtJFrKrUes8MT/KZ3jabD6nbEkHGx+m25c
JhvLhnA6pMoblMlMzWu/8vH/FVCoEqxwUfrjzhY6BlRuqOhWiacOq9CvffltcImy
cc+F7mvd/rB3X6GWJ52N+9S/UDXfSXFZwA9ql7lgyE5DL/fDl+bb7GI+fK8VCHZ
2PolbCtiMF5xoxVu28fdx9r7TcxhdL2VaoIBADmGYfxvgEghALqdW2QmtRRNisWQ
y0/RfED7dNtN8o5vjBCbrov/tQ3Ddbb7a0kwo1NFr1xR7KIki98SkKN0EicrPrfc
+ccs6kASTZcPH/ngG91brOAm9FOG2q5cX6kDKlhgHe+1UYm/34a+2wN0/CwAh7MH
gECABtqx9QCD/DJi+n5ocrYk5RsQJrtnwop4L8X24dRiMiRMis4V9uuyRLQTWV/
k3TOjRgL5eRKbcVwV7c8kmaGDWfM/eVLlQW+wEa0wy+TdsUhlYvgsG5yijkhCAEe
/Az0w5ZulvnLbj5eXKiULWISl0sDCBfJepuINHoUpBwsGzFb7ZXTpk2XlM=
-----END RSA PRIVATE KEY-----

```

**Note:** The public component is usually referred with the extension .pub. This can be extracted from the private key that has both the public and private components. The private keys usually have extension .pem. To generate public key components use ppkfile/spkfile instead of pskfile/sskfile in the above example.

### BIF Example

A sample BIF file, `generate_pem.bif`:

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

### Command

The command to generate keys is, as follows:

```
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 AMD 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`.

```
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:

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

# Versal Authentication Support

Bootgen supports RSA-4096 and ECDSA P384 and P521 curves for Versal adaptive SoC authentication. NIST SHA-3 is used to calculate hash on all partitions/headers. The signature calculated on the hash is placed in the PDI.

**Note:** Unlike Zynq devices and Zynq UltraScale+ MPSoC, for Versal adaptive SoCs, the authentication certificate is placed prior to the partition. The ECDSA P521 curve is not supported for authentication of the bootloader partition (PLM) because the BootROM only supports RSA-4096 or ECDSA-P384 authentication. P521 can, however, be used to authenticate any other partition.

## Meta Header Authentication

For a Versal adaptive SoC, Bootgen authenticates the meta header based on the parameters under the bif attribute "metaheader." A snippet of the usage is shown below.

```
metaheader
{
    authentication = rsa,
    pskfile = psk.pem,
    sskfile = ssk.pem
}
```

## PPK Hash for eFUSE

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

## BIF File Example

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

```
generate_hash_ppk:
{
    pskfile = primary0.pem
    sskfile = secondary0.pem
    image
    {
        name = pmc_ss, id = 0x1c000001
        { type=bootloader, authentication=rsa, file=plm.elf}
        { type=pmcdata, load=0xf2000000, file=pmc_cdo.bin}
    }
}
```

## Command

The command to generate PPK hash for eFUSE programming is:

```
bootgen -image generate_hash_ppk.bif -arch versal -w -o test.bin -
efuseppkbits efuseppksha.txt
```

## Cumulative Secure Boot Operations for Versal Adaptive SoC

Table 36: Cumulative Secure Boot Operations

Boot Type	Operations			Hardware Crypto Engines
	Authentication	Decryption	Integrity (Checksum Verification)	
Non-secure boot	No	No	No	None
Asymmetric Hardware Root-of-Trust (A-HWRoT)	Yes (Required)	No	No	RSA/ECDSA along with SHA3
Symmetric Hardware Root-of-Trust (S-HWRoT) (Forces decryption of PDI with eFUSE black key)	No	Yes (Required PLM and Meta Header should be encrypted with eFUSE KEK)	No	AES-GCM
A-HWRoT + S-HWRoT	Yes (Required)	Yes (Required)	No	RSA/ECDSA along with SHA3 and AES-GCM
Authentication + Decryption of PDI	Yes	Yes (Key source can be either from BBRAM or eFUSE)	No	RSA/ECDSA along with SHA3 and AES-GCM
Decryption (Uses user-selected key. The key source can be of any type such as BBRAM/ BHDR or even eFUSE)	No	Yes	No	AES-GCM
Checksum Verification	No	No	Yes	SHA3

## Versal Hashing Scheme

AMD Versal™ introduces a new hashing scheme that minimizes boot time and buffer space required by the PLM while authenticating partitions. The hashing scheme centers on including the hash for the next block of data in the current block of data (similar to what is done with key rolling). This allows a single signature to be used for the entire partition, regardless of partition size, and removes the need to buffer hashes inside the PLM itself. This scheme is used on all partitions except for the bootloader. This block of data, that is hashed each time, is referred to as secure chunk. This chunk size is 32 KB for Versal.

The hashing scheme as per the table below:



Table 37: Partition Chunking Scheme

Partition Chunk Count	Partition Chunking Scheme	Notes
CHUNK 0	[ Authentication Certificate - Partition Sign Field + SECURE HEADER + GCM TAG + SECURE_CHUNK_SIZE + HASH OF CHUNK 1 ]	This data is hashed and then signed. This signature sits in the Partition Signature field of AC
CHUNK 1	SECURE_CHUNK_SIZE + HASH OF CHUNK 2 ]	
CHUNK 2	SECURE_CHUNK_SIZE + HASH OF CHUNK 3 ]	
CHUNK n-1	SECURE_CHUNK_SIZE + HASH OF CHUNK n]	
CHUNK n	[ REMAINING LENGTH ]	

The SECURE\_CHUNK\_SIZE applicable to AMD Versal™ is 32 KB.

## 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. This module 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 the 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 that 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](#).

## 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.

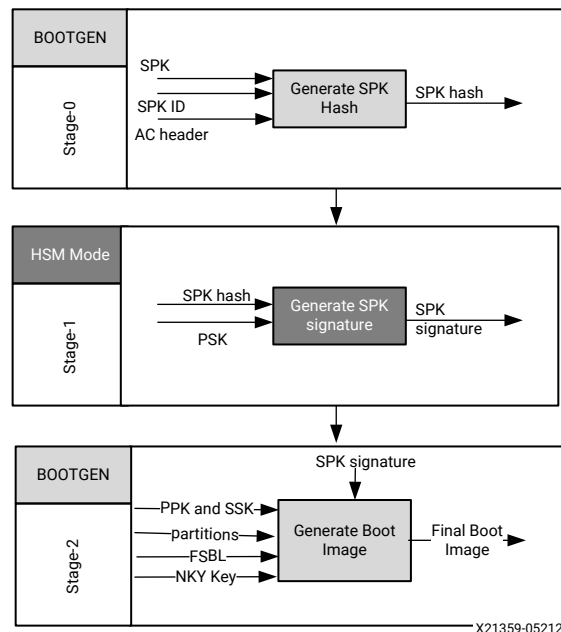
**Note:** HSM flow is not supported for mcs format boot image generation.

## 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 AMD Zynq™ UltraScale+™ MPSoC device to illustrate the stages.

Figure 18: Generic 3-Stage Boot Image



## Boot Process

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

```
all:
{
  [auth_params] ppk_select=1;spk_id=0x8
  [keysrc_encryption]bbam_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 SPK ID. To generate a hash for the above, use the following BIF file snippet.

```

stage 0:
{
    [auth_params] ppk_select=1;spk_id=0x3
    [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 SSK, SPK, and SPKID

```

Stage2:
{
    [keysrc_encryption]bbam_red_key
    [auth_params] ppk_select=1;spk_id=0x3
    [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  
}
```

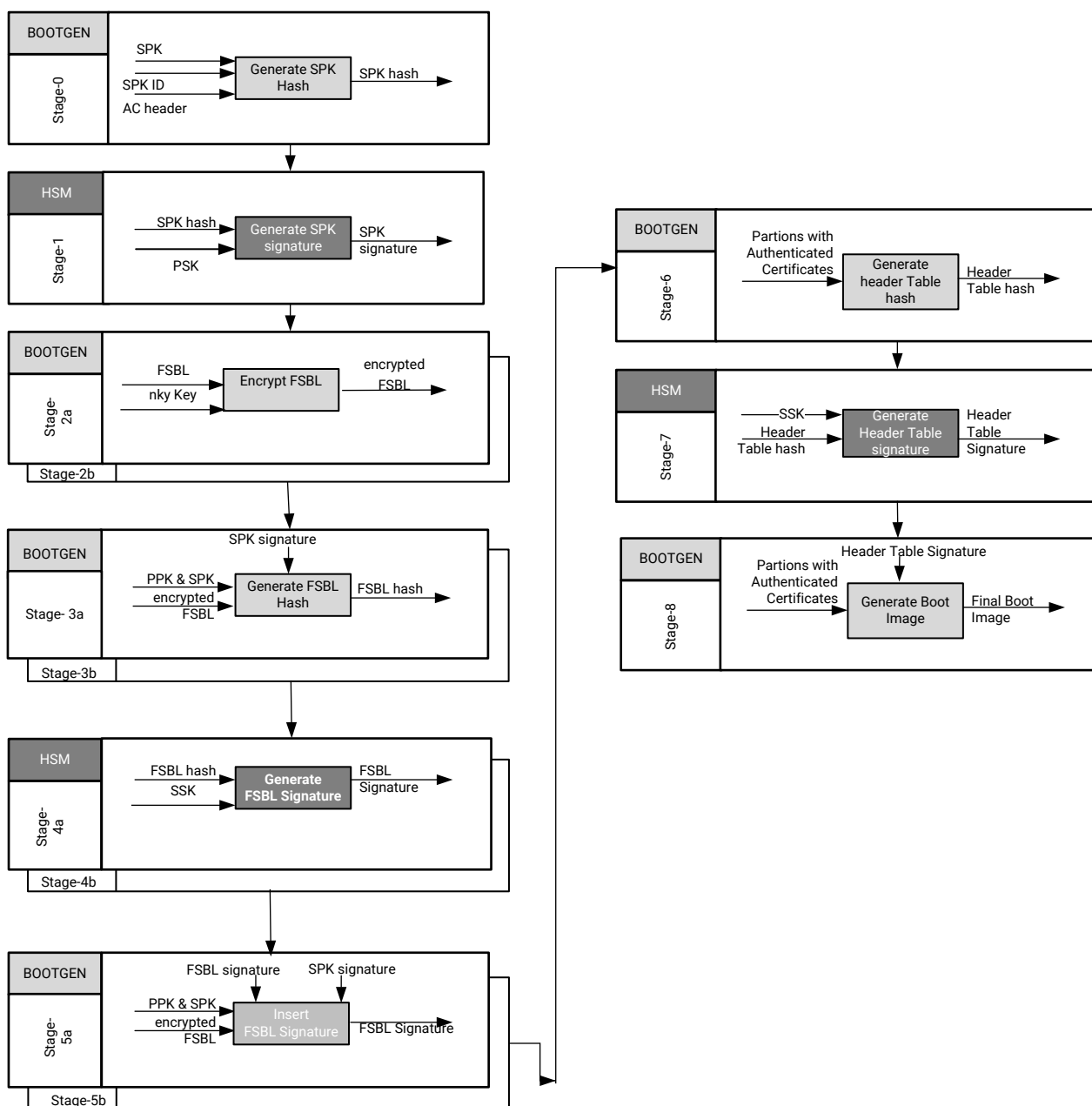
The Bootgen command is:

```
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 an AMD Zynq™ 7000 SoC device. The steps to create this boot image are immediately after the diagram.

Figure 19: Stage 0 to 8 Boot Process



X21416-052120

The process to create a boot image using HSM mode for an AMD 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.

```
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.

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

The following is the Bootgen command.

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

### Stage 1: Sign the SPK Hash

This stage creates the signatures by signing the SPK hash

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

Or by using the following OpenSSL program.

```
#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.

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

The Bootgen command is as follows.

```
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:

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

The Bootgen command is as follows.

```
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:

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

The Bootgen command is as follows.

```
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

```
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.

```
#Swap the bytes in FSBL hash  
objcopy -I binary -O binary --reverse-bytes=256 zynq_fsbl_0.elf.0.sha256  
  
#Generate FSBL signature using OpenSSL  
openssl rsautl -raw -sign -inkey secondary.pem -in zynq_fsbl_0.elf.0.sha256  
> zynq_fsbl_0.elf.0.sha256.sig  
  
#Swap the bytes in FSBL signature  
objcopy -I binary -O binary --reverse-bytes=256 zynq_fsbl_0.elf.0.sha256.sig
```

The output is the signature file `zynq_fsbl_0.elf.0.sha256.sig`.

### Stage 4b: Sign the bitstream hash

```
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.

```
#Swap the bytes in bitstream hash  
objcopy -I binary -O binary --reverse-bytes=256 system.bit.0.sha256  
  
#Generate bitstream signature using OpenSSL  
openssl rsautl -raw -sign -inkey secondary.pem -in system.bit.0.sha256 >  
system.bit.0.sha256.sig  
  
#Swap the bytes in bitstream signature  
objcopy -I binary -O binary --reverse-bytes=256 system.bit.0.sha256.sig
```

The output is the signature file `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 `stage5a.bif` is as follows.

```
stage5a:
{
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature] secondary.pub.sha256.sig
    [bootimage, authentication=rsa, presign=zynq_fsbl_0.elf.0.sha256.sig]
    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:

```
#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:

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

The Bootgen command is as follows:

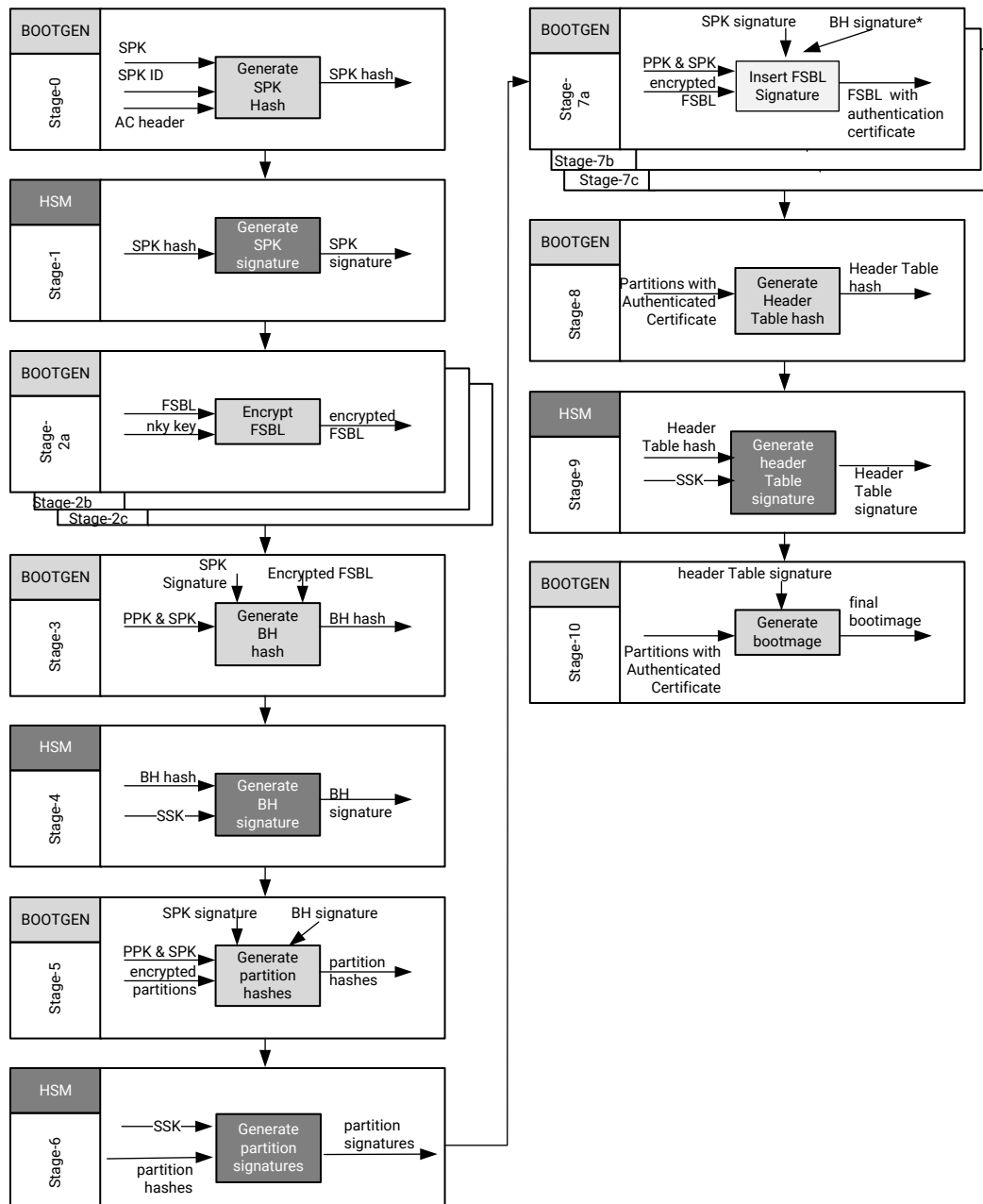
```
bootgen -image stage8.bif -w -o final.bin
```

The output is the boot image 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 20: 0 to 10 Stage Boot Process



X21547-052120

To create a boot image using HSM mode for an AMD 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.

```
all:
{
  [fsbl_config] bh_auth_enable
  [keysrc_encryption] bbam_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
}
```

**Note:** To use pmufw\_image in HSM flow, add [pmufw\_image] pmufw.elf to the above bif. In similar lines, this must be added in the stage2a bif, where FSBL is encrypted. The rest of the flow remains the same.

### Stage 0: Generate a hash for SPK

The following is the snippet from the BIF file.

```
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:

```
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:

```
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:

```
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:

```
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=e1-3,
        trustzone=secure
    ] bl31.elf

    [
        destination_cpu=a53-0,
        authentication=rsa,
        exception_level=e1-2
    ] u-boot.elf
}
```

The Bootgen command is:

```
bootgen -arch zynqmp -image stage5.bif -generate_hashes -w on -log error
```

Multiple hashes are generated for a bitstream partition. For more details, see [Bitstream Authentication Using External Memory](#).

The Boot Header hash is also generated in this stage5; 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 is only used in the HSM mode flow.

## Stage 6: Sign Partition Hashes

Create the following files using OpenSSL:

```
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:

```
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:

```
bootgen -arch zynqmp -image stage7a.bif -o fsbl_e_ac.bin -efuseppkbits
efuseppkbits.txt -nonbooting -w on -log error
```

**Stage 7b:** Insert the bitstream signature by adding the following to the BIF file:

```
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:

```
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:

```
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:

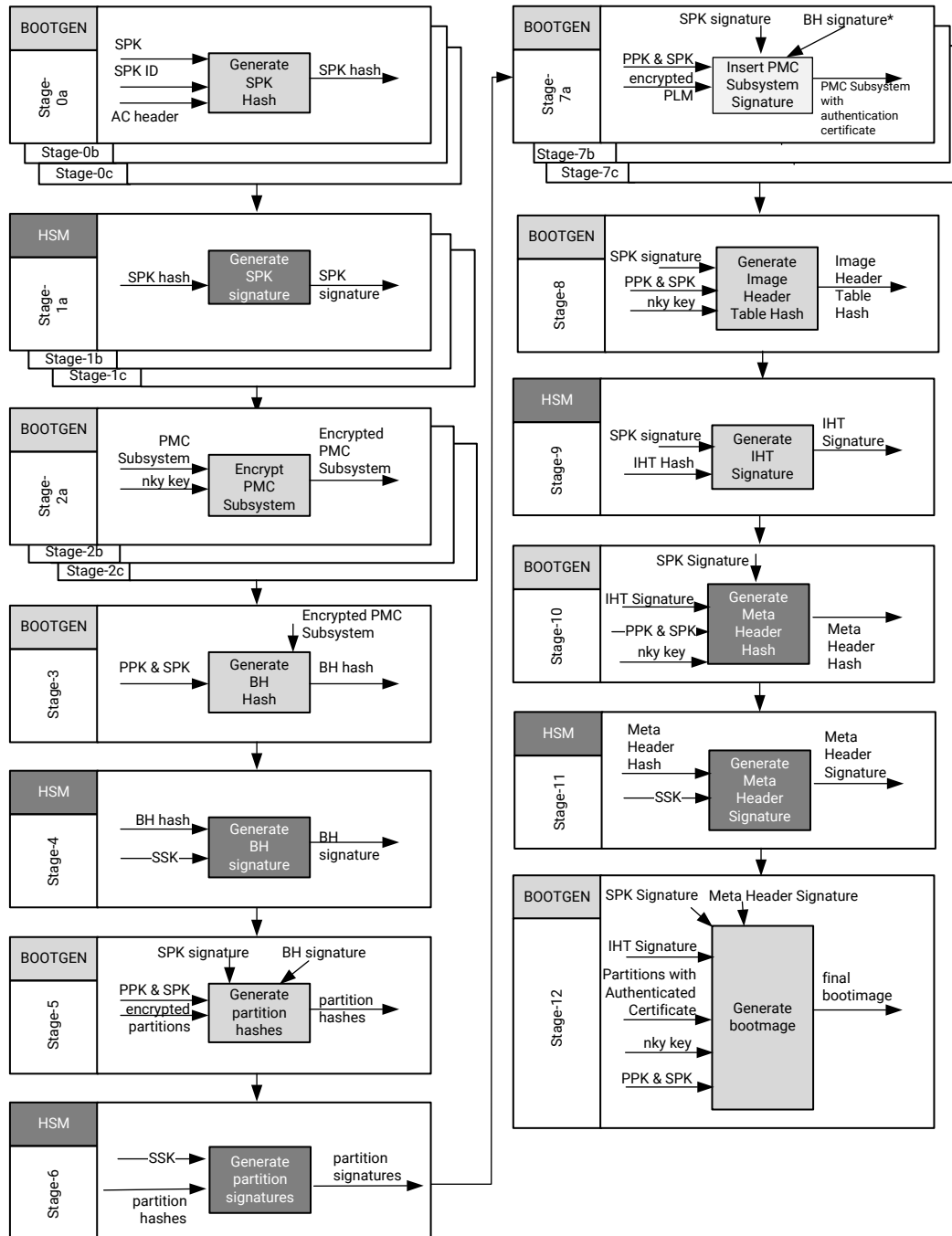
```
bootgen -arch zynqmp -image stage10.bif -o final.bin -w on -log error
```

**Note:** At the moment, there is no support for the HSM mode on Versal devices.

# Creating a Versal Device Boot Image Using HSM

The following figure provides a diagram of an HSM mode boot image for a Versal device.

Figure 21: 0 to 12 Stage Boot Process



X21547-111020

**Note:** The PMC subsystem includes PLM, PMC\_CDO.

## Generating the PDI

Generate the PDI using the standard BIF.

```
command : bootgen -arch versal -image all.bif -w on -o final_ref.bin -log
error

all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    boot_config {bh_auth_enable}

    metaheader
    {
        authentication = rsa,
        pskfile = rsa-keys/PSK2.pem,
        sskfile = rsa-keys/SSK2.pem
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = enc_keys/efuse_red_metaheader_key.nky,
        dpacm_enable
    }

    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition
        {
            id = 0x01, type = bootloader,
            authentication = rsa,
            pskfile = rsa-keys/PSK1.pem,
            sskfile = rsa-keys/SSK1.pem,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = encr_keys/bbam_red_key.nky,
            dpacm_enable,
            file = images/gen_files/plm.elf
        }
        partition
        {
            id = 0x09, type = pmcdata, load = 0xf2000000,
            aeskeyfile = gen_keys/pmcdata.nky,
            file = images/gen_files/pmc_data.cdo
        }
    }

    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0C, type = cdo,
            authentication = rsa,
            pskfile = rsa-keys/PSK3.pem,
            sskfile = rsa-keys/SSK3.pem,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = gen_keys/key1.nky,
            dpacm_enable,
```

```

    file = images/gen_files/lpd_data.cdo
  }
  partition
  {
    id = 0x0B, core = psm,
    authentication = rsa,
    pskeyfile = rsa-keys/PSK1.pem,
    sskfile = rsa-keys/SSK1.pem,
    encryption = aes,
    keysrc = bbram_red_key,
    aeskeyfile = gen_keys/key2.nky,
    dpacm_enable,
    blocks = 8192(20);4096(*),
    file = images/static_files/psm_fw.elf
  }
}

image
{
  name = fpd, id = 0x420c003
  partition
  {
    id = 0x08, type = cdo,
    authentication = rsa,
    pskeyfile = rsa-keys/PSK3.pem,
    sskfile = rsa-keys/SSK3.pem,
    encryption = aes,
    keysrc = bbram_red_key,
    aeskeyfile = gen_keys/key5.nky,
    dpacm_enable,
    file = images/gen_files/fpd_data.cdo
  }
}

image
{
  name = ss, id = 0x1c000033
  partition
  {
    id = 0x0D, type = cdo,
    authentication = rsa,
    pskeyfile = rsa-keys/PSK2.pem,
    sskfile = rsa-keys/SSK2.pem,
    encryption = aes,
    keysrc = bbram_red_key,
    aeskeyfile = gen_keys/key6.nky,
    dpacm_enable,
    file = images/gen_files/subsystem.cdo
  }
}
}

```

## HSM Mode Steps

### Stage 0: Generate SPK Hash

Generate hash for SSK1:

```
command : bootgen -arch versal -image stage0-SSK1.bif -generate_hashes -w
on -log error

stage0-SSK1:
{
    spkfile = rsa-keys/SSK1.pub
}
```

Generate hash for SSK2:

```
command : bootgen -arch versal -image stage0-SSK2.bif -generate_hashes -w
on -log error

stage0-SSK2:
{
    spkfile = rsa-keys/SSK2.pub
}
```

Generate hash for SSK3:

```
command : bootgen -arch versal -image stage0-SSK3.bif -generate_hashes -w
on -log error

stage0-SSK3:
{
    spkfile = rsa-keys/SSK3.pub
}
```

### Stage 1: Sign SPK hash

Sign the generated hashes:

```
openssl rsautl -raw -sign -inkey rsa-keys/PSK1.pem -in SSK1.pub.sha384 >
SSK1.pub.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/PSK2.pem -in SSK2.pub.sha384 >
SSK2.pub.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/PSK3.pem -in SSK3.pub.sha384 >
SSK3.pub.sha384.sig
```

### Stage 2: Encrypt Individual Partitions

Encrypt partition 1:

```
command : bootgen -arch versal -image stage2a.bif -o pmc_subsys_e.bin -w on
-log error

stage2a:
{
    image
    {
```

```

name = pmc_subsys, id = 0x1c000001
partition
{
    id = 0x01, type = bootloader,
    encryption=aes,
    keysrc = bbram_red_key,
    aeskeyfile = encr_keys/bbram_red_key.nky,
    dpacm_enable,
    file = images/gen_files/plm.elf
}
partition
{
    id = 0x09, type = pmcdata,
    load = 0xf2000000,
    aeskeyfile = encr_keys/pmcdata.nky,
    file = images/gen_files/pmc_data.cdo
}
}
}

```

### Encrypt partition 2:

```

command : bootgen -arch versal -image stage2b-1.bif -o lpd_lpd_data-e.bin -w
on -log error

stage2b-1:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0C, type = cdo,
            encryption=aes, delay_auth,
            keysrc = bbram_red_key,
            aeskeyfile = encr_keys/key1.nky,
            dpacm_enable,
            file = images/gen_files/lpd_data.cdo
        }
    }
}

```

### Encrypt partition 3:

```

command : bootgen -arch versal -image stage2b-2.bif -o lpd_psm_fw-e.bin -w
on -log error

stage2b-2:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0B, core = psm,
            encryption = aes, delay_auth,
            keysrc = bbram_red_key,
            aeskeyfile = encr_keys/key2.nky,

```

```

    dpacm_enable,
    file = images/static_files/psm_fw.elf
  }
}
}

```

### Encrypt partition 4:

```

command : bootgen -arch versal -image stage2c.bif -o fpd_e.bin -w on -log
error

stage2c:
{
  image
  {
    name = fpd, id = 0x420c003
    partition
    {
      id = 0x08, type = cdo,
      encryption=aes, delay_auth,
      keysrc = bbam_red_key,
      aeskeyfile = encr_keys/key5.nky,
      dpacm_enable,
      file = images/gen_files/fpd_data.cdo
    }
  }
}

```

### Stage 3: Generate Boot Header Hash

```

command : bootgen -arch versal -image stage3.bif -generate_hashes -w on -
log error

stage3:
{
  image_config {bh_auth_enable}
  image
  {
    name = pmc_subsys, id = 0x1c000001
    {
      type = bootimage,
      authentication=rsa,
      ppkfile = rsa-keys/PSK1.pub,
      spkfile = rsa-keys/SSK1.pub,
      spksignature = SSK1.pub.sha384.sig,
      file = pmc_subsys_e.bin
    }
  }
}

```

### Stage 4: Sign Boot Header Hash

Sign the generated hashes:

```

openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in bootheader.sha384 >
bootheader.sha384.sig

```

**Stage 5: Generate Partition Hashes**

```
command : bootgen -arch versal -image stage5.bif -generate_hashes -w on -log error
```

```
stage5:
{
    bhsignature = bootheader.sha384.sig

    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootimage,
            authentication=rsa,
            ppkfile = rsa-keys/PSK1.pub,
            spkfile = rsa-keys/SSK1.pub,
            spksignature = SSK1.pub.sha384.sig,
            file = pmc_subsys_e.bin
        }
    }

    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            type = bootimage,
            authentication = rsa,
            ppkfile = rsa-keys/PSK3.pub,
            spkfile = rsa-keys/SSK3.pub,
            spksignature = SSK3.pub.sha384.sig,
            file = lpd_lpd_data_e.bin
        }
        partition
        {
            type = bootimage,
            authentication = rsa,
            ppkfile = rsa-keys/PSK1.pub,
            spkfile = rsa-keys/SSK1.pub,
            spksignature = SSK1.pub.sha384.sig,
            file = lpd_psm_fw_e.bin
        }
    }

    image
    {
        id = 0x1c000000, name = fpd
        {
            type = bootimage,
            authentication=rsa,
            ppkfile = rsa-keys/PSK3.pub,
            spkfile = rsa-keys/SSK3.pub,
            spksignature = SSK3.pub.sha384.sig,
            file = fpd_e.bin
        }
    }

    image
    {
        id = 0x1c000033, name = ss
        {
            type = bootimage,
```



```

        authentication = rsa,
        ppkfile = rsa-keys/PSK2.pub,
        spkfile = rsa-keys/SSK2.pub,
        spksignature = SSK2.pub.sha384.sig,
        file = subsystem_e.bin
    }
}
}

```

## Stage 6: Sign Partition Hashes

```

openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in
pmc_subsys_1.0.sha384 > pmc_subsys.0.sha384.sig

openssl rsautl -raw -sign -inkey rsa-keys/SSK3.pem -in lpd_12.0.sha384 >
lpd.0.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in lpd_11.0.sha384 >
psm.0.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in lpd_11.1.sha384 >
psm.1.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in lpd_11.2.sha384 >
psm.2.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in lpd_11.3.sha384 >
psm.3.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/SSK1.pem -in lpd_11.4.sha384 >
psm.4.sha384.sig

openssl rsautl -raw -sign -inkey rsa-keys/SSK3.pem -in fpd_8.0.sha384 >
fpd_data.cdo.0.sha384.sig
openssl rsautl -raw -sign -inkey rsa-keys/SSK2.pem -in ss_13.0.sha384 >
ss.0.sha384.sig

```

## Stage 7: Insert Partition Signatures into Authentication Certificates

Insert partition 1 signature:

```

command : bootgen -arch versal -image stage7a.bif -o pmc_subsys_e_ac.bin -w
on -log error

stage7a:
{
    bhsignature = bootheader.sha384.sig
    image_config {bh_auth_enable}

    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootimage,
            authentication=rsa,
            ppkfile = rsa-keys/PSK1.pub,
            spkfile = rsa-keys/SSK1.pub,
            spksignature = SSK1.pub.sha384.sig,
            presign = pmc_subsys.0.sha384.sig,
            file = pmc_subsys_e.bin
        }
    }
}

```

**Insert partition 2 signature:**

```
command : bootgen -arch versal -image stage7b-1.bif -o
lpd_lpd_data_e_ac.bin -w on -log error

stage7b-1:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            type = bootimage,
            authentication = rsa,
            ppkfile = rsa-keys/PSK3.pub,
            spkfile = rsa-keys/SSK3.pub,
            spksignature = SSK3.pub.sha384.sig,
            presign = lpd.0.sha384.sig,
            file = lpd_lpd_data_e.bin
        }
    }
}
```

**Insert partition 3 signature:**

```
command : bootgen -arch versal -image stage7b-2.bif -o lpd_psm_fw_e_ac.bin -
w on -log error

stage7b-2:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            type = bootimage,
            authentication = rsa,
            ppkfile = rsa-keys/PSK1.pub,
            spkfile = rsa-keys/SSK1.pub,
            spksignature = SSK1.pub.sha384.sig,
            presign = psm.0.sha384.sig,
            file = lpd_psm_fw_e.bin
        }
    }
}
```

**Insert partition 4 signature:**

```
command : bootgen -arch versal -image stage7c.bif -o fpd_e_ac.bin.bin -w on
-log error

stage7c:
{
    image
    {
        id = 0x1c000000, name = fpd
        { type = bootimage,
          authentication=rsa,
          ppkfile = rsa-keys/PSK3.pub,
          spkfile = rsa-keys/SSK3.pub,
          spksignature = SSK3.pub.sha384.sig,
```

```

        presign = fpd_data.cdo.0.sha384.sig,
        file = fpd_e.bin
    }
}

```

### Insert partition 5 signature:

```

command : bootgen -arch versal -image stage7d.bif -o subsystem_e_ac.bin -w
on -log error

stage7d:
{
    image
    {
        id = 0x1c000033, name = ss
        { type = bootimage,
          authentication = rsa,
          ppkfile = rsa-keys/PSK2.pub,
          spkfile = rsa-keys/SSK2.pub,
          spksignature = SSK2.pub.sha384.sig,
          presign = ss.0.sha384.sig,
          file = subsystem_e.bin
        }
    }
}

```

### Stage 8: Generate Image Header Table Hash

```

command : bootgen -arch versal -image stage8a.bif -generate_hashes -w on -
log error

stage8:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    metaheader
    {
        authentication = rsa,
        ppkfile = rsa-keys/PSK2.pub,
        spkfile = rsa-keys/SSK2.pub,
        spksignature = SSK2.pub.sha384.sig,
        encryption=aes,
        keysrc = bbram_red_key,
        aeskeyfile = encr_keys/efuse_red_metaheader_key.nky,
        dpacm_enable,
        revoke_id = 0x00000002
    }

    image
    {
        {type = bootimage, file = pmc_subsys_e_ac.bin}
    }

    image
    {
        {type = bootimage, file = lpd_lpd_data_e_ac.bin}
        {type = bootimage, file = lpd_psm_fw_e_ac.bin}
    }
}

```

```

image
{
    {type = bootimage, file = fpd_e_ac.bin}
}

image
{
    {type = bootimage, file = subsystem_e_ac.bin}
}
}

```

## Stage 9: Sign Image Header Table Hash

Sign the generated hashes:

```

openssl rsautl -raw -sign -inkey rsa-keys/SSK2.pem -in
imageheadertable.sha384 > imageheadertable.sha384.sig

```

## Stage 10: Generate Meta Header Hash

```

command : bootgen -arch versal -image stage8b.bif -generate_hashes -w on -
log error

```

```

stage8b:
{
    headersignature = imageheadertable.sha384.sig
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    metaheader
    {
        authentication = rsa,
        ppkfile = rsa-keys/PSK2.pub,
        spkfile = rsa-keys/SSK2.pub,
        spksignature = SSK2.pub.sha384.sig,
        encryption=aes,
        keysrc = bbram_red_key,
        aeskeyfile = encr_keys/efuse_red_metaheader_key.nky,
        dpacm_enable
    }

    image
    {
        {type = bootimage, file = pmc_subsys_e_ac.bin}
    }

    image
    {
        {type = bootimage, file = lpd_lpd_data_e_ac.bin}
        {type = bootimage, file = lpd_psm_fw_e_ac.bin}
    }

    image
    {
        {type = bootimage, file = fpd_e_ac.bin}
    }
}

```

```

image
{
    {type = bootimage, file = subsystem_e_ac.bin}
}
}

```

## Stage 11: Sign Meta Header Hash

```

openssl rsautl -raw -sign -inkey rsa-keys/SSK2.pem -in MetaHeader.sha384 >
metaheader.sha384.sig

```

## Stage 12: Combine Partitions and Insert Header Signature

Build the complete PDI:

```

command : bootgen -arch versal -image stage10.bif -o final.bin -w on -log
error

stage10:
{
    headersignature = imageheadertable.sha384.sig
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    metaheader
    {
        authentication = rsa,
        ppkfile = rsa-keys/PSK2.pub,
        spkfile = rsa-keys/SSK2.pub
        spksignature = SSK2.pub.sha384.sig,
        presign = metaheader.sha384.sig
        encryption=aes,
        keysrc = bbram_red_key,
        aeskeyfile = encr_keys/efuse_red_metaheader_key.nky,
        dpacm_enable
    }

    image
    {
        {type = bootimage, file = pmc_subsys_e_ac.bin}
    }

    image
    {
        {type = bootimage, file = lpd_lpd_data_e_ac.bin}
        {type = bootimage, file = lpd_psm_fw_e_ac.bin}
    }

    image
    {
        {type = bootimage, file = fpd_e_ac.bin}
    }

    image
    {
        {type = bootimage, file = subsystem_e_ac.bin}
    }
}

```

---

## Creating a Keystone B Device Boot Image Using HSM

The HSM flow for AMD Keystone B series is similar to Versal devices. You can use `-arch versalnet` instead of `-arch versal`.

# SSIT Support

Stacked Silicon Interconnect Technology (SSI technology) is used to break through the limitations of Moore's law and deliver the capabilities to satisfy the most demanding design requirements.

A SSI Technology device consists of multiple super logic region (SLRs), where each SLR is one die. SLR0, also referred to as Master SLR, is the bottom die. SLR1 is second from bottom, SLR2 is third from bottom and so on. The AMD Versal™ adaptive SoC SSI technology variants can have a maximum of four SLRs.

Each SLR has its own platform management controller (PMC) and a programmable logic (PL) region like a monolithic AMD Versal™ adaptive SoC, but the slave SLRs do not have AI Engine and processing system (PS) regions.

The final PDI to boot on this device is a PDI of PDIs. Because each SLR has its own PMC block, each SLR boots with a PDI which is integrated in the main PDI.

**Note:** PLM elf in each SLR should be same.

The BIF for AMD Versal™ adaptive SoCs with SSI technology is different from its monolithic variant. Below is an example bif with two SLR devices.

**Note:** The whole BIF code block below goes into a Single file. Bootgen reads multiple BIF blocks and creates respective PDIs based on the BIF labels. These BIF block labels are referenced in master BIF, based on which Bootgen merges the individual PDIs into a master PDI. This master PDI alone is sufficient to boot an SSI technology device.

**Note:** The Slave SLRs is used the special `smap_width=0` option indicating downstream connection and must not be changed.

## SSI technology Non Secure Bif Example for a Single Slave SLR Device:

## Example- HBM Devices

```

bitstream_boot_1:
{
  id_code = 0x04d28093
  extended_id_code = 0x01
  id = 0xb
  boot_config {smap_width=0}
  image
  {
    name = pmc_subsys
    id = 0x1c000001
    partition
    {
      id = 0xb01
      type = bootloader
      file = gen_files/plm.elf
    }
    partition
    {
      id = 0xb0A
      type = pmcdata, load = 0xf2000000
      file = gen_files/pmc_data_slr_1.cdo
    }
  }
  image
  {
    name = pl_noc
    id = 0x18700000
    partition
    {
      id = 0xb05
      type = cdo
      file = project_1_wrapper_boot_1.rnpi
    }
  }
}

bitstream_1:
{
  id_code = 0x04d28093
  extended_id_code = 0x01
  id = 0xc
  boot_config {smap_width=0}
  image
  {
    name = pl_cfi
    id = 0x18700000
    partition
    {
      id = 0xc03
      type = cdo
      file = project_1_wrapper_1.rcdo
    }
    partition
    {
      id = 0xc05
      type = cdo
      file = project_1_wrapper_1.rnpi
    }
  }
}

```



```

    }
}

bitstream_master:
{
    id_code = 0x04d28093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0x01
            type = bootloader
            slr = 0
            file = gen_files/plm.elf
        }
        partition
        {
            id = 0x09
            type = pmcdata, load = 0xf2000000
            slr = 0
            file = gen_files/pmc_data.cdo
        }
    }
}
image
{
    name = SUB_SYSTEM_BOOT_MASTER
    id = 0x18700000
    type = slr-boot
    partition
    {
        id = 0x05
        type = cdo
        slr = 0
        file = project_1_wrapper_boot_0.rnpi
    }
    partition
    {
        id = 0xb15
        slr = 1
        section = bitstream_boot_1
    }
    partition
    {
        id = 0x02
        type = cdo
        file = noc_pll.rnpi
    }
}
image
{
    name = lpd
    id = 0x4210002
    partition
    {
        id = 0x0C
        type = cdo
        slr = 0
        file = gen_files/lpd_data.cdo
    }
}

```

```

partition
{
    id = 0x0B
    core = psm
    slr = 0
    file = static_files/psm_fw.elf
}
}
image
{
    name = fpd
    id = 0x420c003
    partition
    {
        id = 0x08
        type = cdo
        slr = 0
        file = gen_files/fpd_data.cdo
    }
}
image
{
    name = CONFIG_MASTER
    id = 0x18700000
    type = slr-config
    partition
    {
        id = 0xc16
        slr = 1
        section = bitstream_1
    }
    partition
    {
        id = 0x13
        type = cdo
        slr = 0
        file = project_1_wrapper_master_config.rcdo
    }
}
}
}

```

### SSI technology Non Secure Bif Example for a 2 Slave SLR Device:

```

// For generating SLR1 - boot PDI
bitstream_boot_1:
{
    id_code = 0x04d10093
    extended_id_code = 0x01
    id = 0xb
    boot_config {smap_width=0}
    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition { id = 0xb01, type = bootloader, file = gen_files/plm.elf }
        partition { id = 0xb0A, type = pmcdata, load = 0xf2000000, file =
gen_files/pmc_data_slr_1.cdo }
    }
    image
    {
        name = pl_noc, id = 0x18700000
        partition { id = 0xb05, type = cdo, file = boot_1.rnpi }
    }
}
}

```

```

}

// For generating SLR2 - boot PDI
bitstream_boot_2:
{
    id_code = 0x04d10093
    extended_id_code = 0x01
    id = 0xb
    boot_config {smap_width=0}
    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition { id = 0xb01, type = bootloader, file = gen_files/plm.elf }
        partition { id = 0xb0A, type = pmcdata, load = 0xf2000000, file =
gen_files/pmc_data_slr_2.cdo }
    }
    image
    {
        name = pl_noc, id = 0x18700000
        partition { id = 0xb05, type = cdo, file = boot_2.rnpi }
    }
}

// For generating SLR1 - partial config PDI
bitstream_1:
{
    id_code = 0x04d10093
    extended_id_code = 0x01
    id = 0xc
    boot_config {smap_width=0}
    image
    {
        name = pl_cfi, id = 0x18700000
        partition { id = 0xc03, type = cdo, file = config_1.rcdo }
        partition { id = 0xc05, type = cdo, file = config_1.rnpi }
    }
}

// For generating SLR2 - partial config PDI
bitstream_2:
{
    id_code = 0x04d10093
    extended_id_code = 0x01
    id = 0xc
    boot_config {smap_width=0}
    image
    {
        name = pl_cfi, id = 0x18700000
        partition { id = 0xc03, type = cdo, file = config_2.rcdo }
        partition { id = 0xc05, type = cdo, file = config_2.rnpi }
    }
}

// For generating final PDI - by combining above generated PDIs.
bitstream_master:
{
    id_code = 0x04d10093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition { id = 0x01, type = bootloader, slr = 0, file = gen_files/

```

```

plm.elf }
    partition { id = 0x09, type = pmcdata, load = 0xf2000000, slr = 0, file
= gen_files/pmc_data.cdo }
    }
    image
    {
        name = SUB_SYSTEM_BOOT_MASTER, id = 0x18700000, type = slr-boot
        partition { id = 0x05, type = cdo, slr = 0, file = boot_0.rnpi }
        partition { id = 0xb15, slr = 1, section = bitstream_boot_1 }
        partition { id = 0xb15, slr = 2, section = bitstream_boot_2 }
        partition { id = 0x02, type = cdo, file = gen_files/
bd_70da_pspmc_0_0_noc_clock.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        partition { id = 0x0C, type = cdo, slr = 0, file = gen_files/
lpd_data.cdo }
        partition { id = 0x0B, core = psm, slr = 0, file = static_files/
psm_fw.elf }
    }
    image
    {
        name = fpd, id = 0x420c003
        partition { id = 0x08, type = cdo, slr = 0, file = gen_files/
fpd_data.cdo }
    }
    image
    {
        name = CONFIG_MASTER, id = 0x18700001, type = slr-config
        partition { id = 0xc16, slr = 1, section = bitstream_1 }
        partition { id = 0xc16, slr = 2, section = bitstream_2 }
        partition { id = 0x13, type = cdo, slr = 0, file = master_config.rcdo }
    }
}

```

**Note:** MCS format bootimage/PDI generation is supported for SSI technology devices. The intermediate PDIs that are generated for slaves are always in binary format. The final PDI generated would be in mcs format, if chosen.

#### SSI technology Authentication Bif Example for a 3 Slave SLR Device:

```

command : bootgen -arch versal -image all.bif -w on -o final_ref.bin -log
error

bitstream_boot_1:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0xb

    boot_config {smap_width=0,bh_auth_enable}
    pskeyfile = PSK1.pem
    sskfile = SSK1.pem

    metaheader
    {
        authentication = rsa
    }
    image
    {
        name = pmc_subsys

```

```

    id = 0x1c000001
    partition
    {
        id = 0xb01
        type = bootloader
        authentication = rsa
        file = gen_files/plm.elf
    }
    partition
    {
        id = 0xb0A
        type = pmcdata, load = 0xf2000000
        file = gen_files/pmc_data_slr_1.cdo
    }
}
image
{
    name = pl_noc
    id = 0x18700000
    partition
    {
        id = 0xb05
        type = cdo
        authentication = rsa
        file = project_1_wrapper_boot_1.rnpi
    }
}

bitstream_boot_2:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0xb

    boot_config {smap_width=0,bh_auth_enable}
    pskfile = PSK2.pem
    sskfile = SSK2.pem

    metaheader
    {
        authentication = rsa
    }
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0xb01
            type = bootloader
            authentication = rsa
            file = gen_files/plm.elf
        }
        partition
        {
            id = 0xb0A
            type = pmcdata, load = 0xf2000000
            file = gen_files/pmc_data_slr_2.cdo
        }
    }
}
image
{

```

```

    name = pl_noc
    id = 0x18700000
    partition
    {
        id = 0xb05
        type = cdo
        authentication = rsa
        file = project_1_wrapper_boot_2.rnpi
    }
}

bitstream_boot_3:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0xb

    boot_config {smap_width=0,bh_auth_enable}
    pskfile = PSK3.pem
    sskfile = SSK3.pem

    metaheader
    {
        authentication = rsa
    }
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0xb01
            type = bootloader
            authentication = rsa
            file = gen_files/plm.elf
        }
        partition
        {
            id = 0xb0A
            type = pmcdata, load = 0xf2000000
            file = gen_files/pmc_data_slr_3.cdo
        }
    }
    image
    {
        name = pl_noc
        id = 0x18700000
        partition
        {
            id = 0xb05
            type = cdo
            authentication = rsa
            file = project_1_wrapper_boot_3.rnpi
        }
    }
}

bitstream_1:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0xc

```

```
pskfile = PSK1.pem
sskfile = SSK1.pem
boot_config {smap_width=0,bh_auth_enable}

metaheader
{
  authentication = rsa
}
image
{
  name = pl_cfi
  id = 0x18700000
  partition
  {
    id = 0xc03
    type = cdo
    authentication = rsa
    file = project_1_wrapper_1.rcdo
  }
  partition
  {
    id = 0xc05
    type = cdo
    authentication = rsa
    file = project_1_wrapper_1.rnpi
  }
}

bitstream_2:
{
  id_code = 0x04d14093
  extended_id_code = 0x01
  id = 0xc

  pskfile = PSK2.pem
  sskfile = SSK2.pem
  boot_config {smap_width=0,bh_auth_enable}

  metaheader
  {
    authentication = rsa
  }
  image
  {
    name = pl_cfi
    id = 0x18700000
    partition
    {
      id = 0xc03
      type = cdo
      authentication = rsa
      file = project_1_wrapper_2.rcdo
    }
    partition
    {
      id = 0xc05
      type = cdo
      authentication = rsa
      file = project_1_wrapper_2.rnpi
    }
  }
}
```

```

}

bitstream_3:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0xc

    pskfile = PSK3.pem
    sskfile = SSK3.pem
    boot_config {smap_width=0,bh_auth_enable}

    metaheader
    {
        authentication = rsa
    }
    image
    {
        name = pl_cfi
        id = 0x18700000
        partition
        {
            id = 0xc03
            type = cdo
            authentication = rsa
            file = project_1_wrapper_3.rcdo
        }
        partition
        {
            id = 0xc05
            type = cdo
            authentication = rsa
            file = project_1_wrapper_3.rnpi
        }
    }
}

bitstream_master:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0x2

    boot_config { bh_auth_enable }
    pskfile = psk.pem
    sskfile = ssk.pem

    metaheader
    {
        authentication = rsa
    }
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0x01
            type = bootloader
            slr = 0
            authentication = rsa
            file = gen_files/plm.elf
        }
    }
}

```



```
partition
{
    id = 0x09
    type = pmcdata, load = 0xf2000000
    slr = 0
    file = gen_files/pmc_data.cdo
}
}
image
{
    name = BOOT_MAS_AUTH
    id = 0x18700000
    type = slr-boot
    partition
    {
        id = 0x05
        type = cdo
        slr = 0
        authentication = rsa
        file = project_1_wrapper_boot_0.rnpi
    }
    partition
    {
        id = 0xb15
        slr = 1
        authentication = rsa
        section = bitstream_boot_1
    }
    partition
    {
        id = 0xb15
        slr = 2
        authentication = rsa
        section = bitstream_boot_2
    }
    partition
    {
        id = 0xb15
        slr = 3
        authentication = rsa
        section = bitstream_boot_3
    }
    partition
    {
        id = 0x02
        type = cdo
        authentication = rsa
        file = noc_pll.rnpi
    }
}
}
image
{
    name = lpd
    id = 0x4210002
    partition
    {
        id = 0x0C
        type = cdo
        slr = 0
        authentication = rsa
        file = gen_files/lpd_data.cdo
    }
    partition
```

```
{
  id = 0x0B
  core = psm
  slr = 0
  authentication = rsa
  file = static_files/psm_fw.elf
}
}
image
{
  name = fpd
  id = 0x420c003
  partition
  {
    id = 0x08
    type = cdo
    slr = 0
    authentication = rsa
    file = gen_files/fpd_data.cdo
  }
}
image
{
  name = CONF_MAS_AUTH
  id = 0x18700000
  type = slr-config
  partition
  {
    id = 0xc16
    slr = 1
    authentication = rsa
    section = bitstream_1
  }
  partition
  {
    id = 0xc16
    slr = 2
    authentication = rsa
    section = bitstream_2
  }
  partition
  {
    id = 0xc16
    slr = 3
    authentication = rsa
    section = bitstream_3
  }
  partition
  {
    id = 0x13
    type = cdo
    slr = 0
    authentication = rsa
    file = project_1_wrapper_master_config.rcdo
  }
}
```

# 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. AMD 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

AMD 7 series 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. For UltraScale devices and beyond, AES-256/Galois Counter Mode (GCM) are used, and HMAC is not required.

### Encryption Example

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

```
bootgen -arch fpga -image secure.bif -w -o securetop.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:

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

## Family or Obfuscated Key

To support obfuscated key encryption, you must register with AMD 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](mailto:secure.solutions@xilinx.com) to obtain the Family Key.

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

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

Figure 22: AES Key Sample

```
Device xcku115;
EncryptKeySelect BBRAM;
KeyObfuscate 94da9014cb2203f502f81d14fa2471f4a8902b16d9d408c9c66db214c1640db7, 0;
StartIvObfuscate c485144e397a92081ad20c867a005272, 0;
Key0 dcd2e72ad1b281ecca5e0790b65b94090ec1c8fc010eb01e56717345df4c7010, 0;
StartIV0 3fe826e5495db1bdaf0c2ca2e8640911, 0;
KeyObfuscate 967a6dlecccefddd1990241007de18f41d69ca7231852c0061fb6c78e204c5f3, 1;
StartIvObfuscate 7ab9a7ca88474d7f95ed1b548523451b, 1;
Key0 af84947a9cc256c090d5aefc53ed3fd33bb553d7039e445829ba4cffbe56ffe3, 1;
StartIV0 a50026e212363e1d71fa6f4fb540ce42, 1;
```

## HSM Mode

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

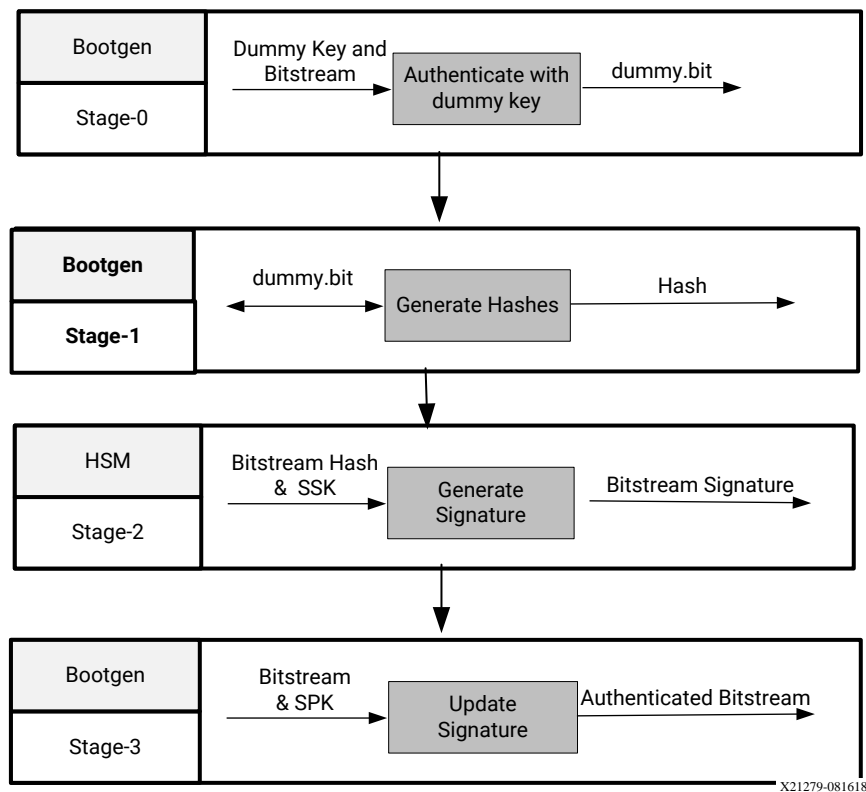
## Standard Mode

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:

```
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 23: 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.

```
bootgen -arch fpga -image stage0.bif -w -o dummy.bit -log error
```

The content of `stage0.bif` is as follows. Refer to the next stages for format.

```
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
```

`Stage1.bif` is as follows:

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

### Stage 2: Sign the hash HSM

Here, OpenSSL is used for demonstration.

```
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:

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

```
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.

# HSM Flow with Both Authentication and Encryption

**Stage 0: Encrypt and authenticate the plain bitstream with dummy key. Add the keylife parameter if keyrolling is required.**

You can provide the `.nky` file, or Bootgen can generate `.nky` file that contains the keys for encryption. Obfuscated AES key generation is not supported by Bootgen. The `keylife` parameter is necessary for the keyrolling feature.

```
the_ROM_image:
{
[aeskeyfile] encrypt.nky
[sskfile] dummykey.pem
[encryption=aes, authentication=rsa,keylife =32] plain-system.bit
}

bootgen -arch fpga -image stage0.bif -w -o auth-encrypt-system.bit -log info
```

After this step, the `.nky` file is generated if encryption is enabled. This file contains all the keys.

## Stage 1: Generate hashes

See the following code for an example.

```
the_ROM_image:
{
[authentication=rsa] auth-encrypt-system.bit
}

bootgen -arch fpga -image stage1.bif -generate_hashes -log info
```

## Stage 2: Sign the hash HSM

Here, OpenSSL is used for demonstration.

```
openssl rsautl -sign -inkey rsaPrivKeyInfo.pem -in auth-encrypt-
system.sha384 > auth-encrypt-system.sha384.sig
```

You can use the HSM server to sign the hashes. For SSI technology devices, generate the signatures for each super logic region (SLR). The following example shows the code to generate the signatures for a device with four SLRs.

```
openssl rsautl -sign -inkey rsaPrivKeyInfo.pem -in auth-encrypt-  
system.0.sha384 > auth-encrypt-system.0.sha384.sig  
  
openssl rsautl -sign -inkey rsaPrivKeyInfo.pem -in auth-encrypt-  
system.1.sha384 > auth-encrypt-system.1.sha384.sig  
  
openssl rsautl -sign -inkey rsaPrivKeyInfo.pem -in auth-encrypt-  
system.2.sha384 > auth-encrypt-system.2.sha384.sig  
  
openssl rsautl -sign -inkey rsaPrivKeyInfo.pem -in auth-encrypt-  
system.3.sha384 > auth-encrypt-system.3.sha384.sig
```

### Stage 3: Update the RSA certificate with the actual signature

See the following code for an example.

```
the_ROM_image:  
{  
[spkfile] rsaPubKeyInfo.pem  
  
[authentication=rsa, presign=auth-encrypt-system.sha384.sig] auth-encrypt-  
system.bit  
}  
  
Command:bootgen -arch fpga -image stage3.bif -w -o rsa-encrypt.bit -log info
```

**Note:** For SSI technology devices, use `presign=<first presign filename>:<number of total presigns>`. For example, a device with four SLRs should have `<first presign filename:4>`.



# 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
}
```

### PMU Firmware 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
}
```

This example shows how to create a boot image with `pmu_fw.elf` loaded by BootROM. If PMU firmware is specified with attribute `[pmufw_image]`, then PMU firmware is not treated as a separate partition. It is appended to the FSBL, FSBL and PMU firmware together becomes one single large partition. Hence, you cannot not see the PMU firmware in the Bootgen log as well.

## PMU Firmware Load by FSBL

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

```
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 an AMD 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`.

```
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:

```
the_ROM_image:
{
    [keysrc_encryption] bbram_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: 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.

```

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.

```

the_ROM_image:
{
    [fsbl_config] puf4kmode, shutter=0x0100005E
    [auth_params] ppk_select=0; spk_id=0x5
    [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.

```
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
    [keysrc_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.

```
the_ROM_image:
{
    [keysrc_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 `bh_gry_key` stored in the Boot Header.

```
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
}
```

## Operational Key

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

```
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
}
```

## 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-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) and 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, and 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
  [keysrc_encryption] bbam_red_key
  [
    bootloader,
    destination_cpu=a53-0,
    encryption=aes,aeskeyfile=aes.nky
  ] fsbl.elf
}
```

Example `aes.nky` for stage1:

```
Device xc7z020clg484;
Key 0 AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
IV 0 F7F8FDE08674A28DC6ED8E37;
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
```

#### 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:**

```
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:**

```
stage1:
{
    [fsbl_config] opt_key
    [keysrc_encryption] bbam_red_key

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

**Example aes.nky for stage1:**

```
Device xc7z020clg484;
Key 0 AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
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:

```
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:

```
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 F7F8FDE08674A28DC6ED8E37;
```

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

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

Example of stage3.bif:

```
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

```
zynqmp 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=0x1
    [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 an error code in the event of a failure. If `RSA_EN` eFUSE is programmed, then 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

    [
        bootloader,
        authentication=rsa,
        destination_cpu=a53-0
    ] ZynqMP_Fsbl.elf

    [destination_cpu=a53-0, encryption=aes] App_A53_0.elf
}
```

## 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 efuse authentication. This is the default selection. In this process the PPK hash in the boot image is verified with the hash from efuse.

```
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, authentication=aes] App_A53_0.elf
}
```

## XIP

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

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

See [xip\\_mode](#) for more information about the command.

## Split with "Offset" Attribute

This example helps to understand how split works with offset attribute.

```
the_ROM_image:
{
  [split]mode=slaveboot,fmt=bin
  [bootloader, destination_cpu = a53-0] fsbl.elf
  [destination_cpu = pmu, offset=0x3000000] pmufw.elf
  [destination_device = pl, offset=0x4000000] design_1_wrapper.bit
  [destination_cpu = a53-0, exception_level = el-3, trustzone,
  offset=0x60000000]\ hello.elf
}
```

When offset is specified to a partition, then the address of that partition in the boot image starts from the given offset. To cover any gap between the mentioned offset of the current partition and the previous partition, bootgen appends 0xFFs to the previous partition. So, now when split is tried on the same, the boot image is expected to be split based on the address of that partition, which is the mentioned offset in this case. So, you see the padded 0xFFs in the split partition outputs.

---

## Versal Adaptive SoC Use Cases

For AMD Versal™ adaptive SoC, AMD Vivado™ generates a boot image known as programmable device image (PDI). This AMD Vivado™ generated PDI contains the bootloader software executable – PLM, along with PL related components, and supporting data files. Based on the project and the CIPS configuration, Vivado creates a BIF file and invokes Bootgen to create the PDI. This BIF is exported as part of XSA to software tools like AMD Vitis™. The BIF can then be modified with required partitions and attributes. Ensure that the lines related to `id_code` and `extended_id_code` are retained as is in the BIF file. This information is mandatory for the PDI image generation by Bootgen.

If you want to write the BIF manually, refer to the BIF generated by Vivado for the same device and ensure that the lines related to `id_code` and `extended_id_code` are added to the BIF that you are writing manually. The sample BIF generated by Vivado is as follows:

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0x01
            type = bootloader
            file = gen_files/plm.elf
        }
        partition
        {
            id = 0x09
            type = pmcdata, load = 0xf2000000
            file = gen_files/pmc_data.cdo
        }
    }
    image
    {
        name = lpd
        id = 0x4210002
        partition
        {
            id = 0x0C
            type = cdo
            file = gen_files/lpd_data.cdo
        }
        partition
        {
            id = 0x0B
            core = psm
            file = static_files/psm_fw.elf
        }
    }
    image
    {
        name = pl_cfi
        id = 0x18700000
        partition
        {
            id = 0x03
            type = cdo
            file = system.rcdo
        }
        partition
        {
            id = 0x05
            type = cdo
            file = system.rnpi
        }
    }
    image
    {
```

```

        name = fpd
        id = 0x420c003
        partition
        {
            id = 0x08
            type = cdo
            file = gen_files/fpd_data.cdo
        }
    }
}

```

**Note:** The BIF file generated in a Vivado project is located in `<vivado_project>/<vivado_project>.runs/impl_1/<Vivado_project>-wrapper.pdi.bif`.

## Bootloader, PMC\_CDO

This example shows how to use Bootloader with PMC\_CDO.

```

all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01

    init = reginit.ini
    image
    {
        {type=bootloader, file=PLM.elf}
        {type=pmcdata, file=pmc_cdo.bin}
    }
}

```

## Bootloader, PMC\_CDO with Load Address

This example shows how to use Bootloader with PMC\_CDO and load address.

```

all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01

    init = reginit.ini
    image
    {
        {type=bootloader, file=PLM.elf}
        {type=pmcdata, load=0xf0400000, file=pmc_cdo.bin}
    }
}

```

## Enable Checksum for Bootloader

This example shows how to enable checksum while using bootloader.

```
all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01

    init = reginit.ini
    image
    {
        {type=bootloader, checksum=sha3, file=PLM.elf}
        {type=pmcdata, load=0xf0400000, file=pmc_cdo.bin}
    }
}
```

## Bootloader, PMC\_CDO, PL CDO, NPI

This example shows how to use bootloader with PMC\_CDO and NPI.

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { id = 0x01, type = bootloader, file = gen_files/plm.elf }
        { id = 0x09, type = pmcdata, load = 0xf2000000, file = gen_files/
pmc_data.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        { id = 0x0C, type = cdo, file = gen_files/lpd_data.cdo }
        { id = 0x0B, core = psm, file = static_files/psm_fw.elf }
    }
    image
    {
        name = pl_cfi, id = 0x18700000
        { id = 0x03, type = cdo, file = system.rcdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = gen_files/fpd_data.cdo }
    }
}
```

## Bootloader, PMC\_CDO, PL CDO, NPI, PS CDO, and PS ELFs

This example shows how to use bootloader with PMC\_CDO, NPI, PS CDO, and PS ELFs.

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { id = 0x01, type = bootloader, file = gen_files/plm.elf }
        { id = 0x09, type = pmcdata, load = 0xf2000000, file = gen_files/
pmc_data.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        { id = 0x0C, type = cdo, file = gen_files/lpd_data.cdo }
        { id = 0x0B, core = psm, file = static_files/psm_fw.elf }
    }
    image
    {
        name = pl_cfi, id = 0x18700000
        { id = 0x03, type = cdo, file = system.rcdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = gen_files/fpd_data.cdo }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core = a72-0, file = apu.elf }
        { core = r5-0, file = rpu.elf }
    }
}
```

## AI Engine Configuration and AI Engine Partitions

This example shows how to configure an AI Engine boot image and AI Engine partitions.

```
all:
{
    image
    {
        { type=bootimage, file=base.pdi }
    }
    image
    {
        name=default_subsys, id=0x1c000000
        { type=cdo
            file = Work/ps/cdo/aie.cdo.reset.bin
            file = Work/ps/cdo/aie.cdo.clock.gating.bin
        }
    }
}
```

```

        file = Work/ps/cdo/aie.cdo.error.handling.bin
        file = Work/ps/cdo/aie.cdo.elfs.bin
        file = Work/ps/cdo/aie.cdo.init.bin
        file = Work/ps/cdo/aie.cdo.enable.bin
    }
}
}

```

**Note:** The different CDOs are merged to form a single partition in the PDI.

## Appending New Partitions to Existing PDI

This example shows how to append new partitions to an existing PDI.

1. Take a Vivado generated PDI (`base.pdi`).
2. Create a new PDI by appending the dtb, uboot, and bl31 applications.

```

new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { load = 0x1000, file = system.dtb }
        { exception_level = el-2, file = u-boot.elf }
        { core = a72-0, exception_level = el-3, trustzone, file =
bl31.elf }
    }
}

```

## RSA Authentication Example

This example demonstrates the use of RSA authentication.

```

all:
{
    id_code = 0x04CA8093
    extended_id_code = 0x01
    boot_config {bh_auth_enable}
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {type = bootloader,
        authentication=rsa, pskfile = ./PSK.pem, sskfile = ./SSK2.pem, revoke_id = 0x2,
        file = ./plm.elf}
        {type = pmcdata, file = ./pmc_data.cdo}
    }
    metaheader
    {
        authentication=rsa,pskfile = ./PSK.pem, sskfile = ./SSK16.pem, revoke_id = 0x10,
    }
    image
    {
        name = lpd, id = 0x4210002
        {type = cdo,

```



```

        authentication=rsa, pskeyfile = ./PSK1.pem, sskfile = ./SSK1.pem, revoke_id = 0x1,
        file = ./lpd_data.cdo}
    { core = psm, file = ./psm_fw.elf}
}
image
{
    name = fpd, id = 0x420c003
    {type = cdo,
    authentication=rsa, pskeyfile = ./PSK1.pem, sskfile = ./SSK5.pem, revoke_id = 0x5,
    file = ./fpd_data.cdo}
}
}

```

## ECDSA Authentication Example

This example demonstrates the use of ECDSA authentication.

```

all:
{
    id_code = 0x04CA8093
    extended_id_code = 0x01
    boot_config {bh_auth_enable}
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {type = bootloader,
        authentication = ecdsa-p384, pskeyfile = ./PSK.pem, sskfile = ./SSK2.pem, revoke_id
= 0x2, file = ./plm.elf}
        {type = pmcdata, file = ./pmc_data.cdo}
    }
    metaheader
    {
        authentication = ecdsa-p384,pskeyfile = ./PSK.pem, sskfile = ./SSK16.pem, revoke_id
= 0x10,
    }
    image
    {
        name = lpd, id = 0x4210002
        {type = cdo,
        authentication = ecdsa-p521, pskeyfile = ./PSK1.pem, sskfile = ./SSK1.pem, revoke_id
= 0x1, file = ./lpd_data.cdo}
        { core = psm, file = ./psm_fw.elf}
    }
    image
    {
        name = fpd, id = 0x420c003
        {type = cdo,
        authentication = ecdsa-p384, pskeyfile = ./PSK1.pem, sskfile = ./SSK5.pem, revoke_id
= 0x5, file = ./fpd_data.cdo}
    }
}

```

## AES Encryption Example

This example demonstrates the use of AES Encryption.

```

all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01

    image
    {

```

```

        {type=bootloader, encryption=aes, keysrc=bbam_red_key, aeskeyfile=key1.nky,
file=plm.elf}
        {type=pmcdata, load=0xf0400000, file=pmc_cdo.bin}
    }
}

```

## AES Encryption with Key Rolling Example

This example demonstrates the use of AES Encryption with key rolling.

```

all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01

    image
    {
        {
            type=bootloader,
            encryption=aes,
            keysrc=bbam_red_key,
            aeskeyfile=key1.nky,
            blocks=65536;32768;16384;8192;4096;2048;1024;512,
            file=plm.elf
        }
        {
            type=pmcdata,
            load=0xf0400000,
            file=pmc_cdo.bin
        }
    }
}

```

## AES Encryption with Multiple Key Sources Example

This example demonstrates the use of different key sources for different partitions.

```

all:
{
    bh_keyfile = ./PUF4K_KEY.txt
    puf_file = ./PUFHD_4K.txt
    bh_kek_iv = ./blk_iv.txt
    bbam_kek_iv = ./bbam_blkIv.txt
    efuse_kek_iv = ./efuse_blkIv.txt
    boot_config {puf4kmode , shutter=0x8100005E}
    id_code = 0x04CA8093
    extended_id_code = 0x01
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {type = bootloader,
            encryption = aes, keysrc=bbam_blk_key, dpacm_enable, revoke_id = 0x5, aeskeyfile
= ./plm.nky, file = ./plm.elf}
        {type = pmcdata,
            aeskeyfile = pmcCdo.nky,
            file = ./pmc_data.cdo}
        }
    metaheader
    {
        encryption = aes, keysrc=bbam_blk_key, dpacm_enable, revoke_id = 0x6,
        aeskeyfile = metaheader.nky
    }
}
image

```

```

{
    name = lpd, id = 0x4210002
    {type = cdo,
    encryption = aes, keysrc = bh_blk_key, pufhd_bh, revoke_id = 0x8, aeskeyfile = ./
lpd.nky, file = ./lpd_data.cdo}
    { core = psm, file = ./psm_fw.elf}
}
image
{
    name = fpd, id = 0x420c003
    {type = cdo,
    encryption = aes, keysrc = efuse_blk_key, dpacm_enable, revoke_id = 0x10, aeskeyfile
= ./fpdcdo.nky, /*Here PUF helper data is also on efuse */ file = ./fpd_data.cdo}
    }
}

```

## AES Encryption and Authentication Example

This example demonstrates the use of AES encryption and authentication.

```

all:
{
    bh_kek_iv = ./blkiv.txt
    bh_keyfile = ./blkkey.txt
    efuse_kek_iv = ./efuse_blkIv.txt
    boot_config {bh_auth_enable, puf4kmode , shutter=0x8100005E}
    id_code = 0x04CA8093
    extended_id_code = 0x01
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {type = bootloader,
        encryption = aes, keysrc=bh_blk_key, dpacm_enable, revoke_id = 0x5, aeskeyfile = ./
plm.nky, authentication = rsa, pskeyfile = ./PSK1.pem, sskfile = ./SSK5.pem,
        file = ./plm.elf}
        {type = pmcdata, aeskeyfile = ./pmc_data.nky, file = ./pmc_data.cdo}
    }
    metaheader
    {
        encryption = aes, keysrc=bh_blk_key, dpacm_enable, revoke_id = 0x6,
        aeskeyfile = metaheader.nky
    }
    image
    {
        name = lpd, id = 0x4210002
        {type = cdo,
        encryption = aes, keysrc = bbram_red_key, revoke_id = 0x8, aeskeyfile = lpd.nky,
        file = ./lpd_data.cdo}
        { core = psm, file = ./psm_fw.elf}
    }
    image
    {
        name = fpd, id = 0x420c003
        {type = cdo,
        encryption = aes, keysrc = efuse_blk_key, dpacm_enable, revoke_id = 0x10,
aeskeyfile = fpd.nky, authentication = ecdsa-p384, pskeyfile = ./PSK1.pem, sskfile = ./
SSK5.pem,
        file = ./fpd_data.cdo}
    }
}

```

## Replacing PLM from an Existing PDI

This example shows the steps to replacing PLM from an existing PDI.

1. Take a Vivado generated PDI (`base.pdi`).
2. Create a new PDI by replacing the PLM (bootloader) from the base PDI.

```
new_bif:
{
  image
  {
    { type = bootimage, file = base.pdi }
    { type = bootloader, file = plm_v1.elf }
  }
}
```

Bootgen replaces the bootloader `plm.elf` with a new `plm_v1.elf`.

### Example Bootgen Command to Create a PDI

Use the following command to create a PDI.

```
bootgen -arch versal -image filename.bif -w -o boot.pdi
```

## Replace PLM and PMC CDO in SSI technology PDIs

Bootgen supports replacing PLM, PMC CDO, and PSM for Monolithic devices, by taking the PDI as input.

Keeping in view the SSI technology devices and future requirements, the replace functionality is now done by taking the bif as input, instead of boot image.

This example shows the steps to replace PLM and PMC DATA.

1. Take a Vivado generated bif (`base.bif`).
2. Create a new PDI by replacing the PLM (bootloader) from the base bif.

```
include: base.bif
replace_image:
{
  image
  {
    partition { type = bootloader, slr = 0, file = plm_v1.elf }
    partition { type = pmcdata, slr = 0, file = pmc_data-v1.cdo }
  }
  image
  {
    partition { type = bootloader, slr = 1, file = plm_v1.elf }
    partition { type = pmcdata, slr = 1, file = pmc_data-v1.cdo }
  }
  image
  {
    partition { type = bootloader, slr = 2, file = plm_v1.elf }
    partition { type = pmcdata, slr = 2, file = pmc_data-v1.cdo }
  }
}
```

Bootgen replaces respective slr bootloader plm.elf with a new plm\_v1.elf and pmcdata with pmc\_data-v1.cdcs.

### Example Bootgen Command to Create a PDI

Use the following command to create a PDI.

```
bootgen -arch versal -image filename.bif -w -o boot.pdi
```

# BIF Attribute Reference

---

## aarch32\_mode

### Syntax

- For AMD Zynq™ UltraScale+™ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{aarch32_mode, file=<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

- For Zynq UltraScale+ MPSoC:

```
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
}
```

- For Versal adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core = a72-0, aarch32_mode, file = apu.bin }
    }
}
```

**Note:** \*base.pdi is the PDI generated by Vivado.

---

## aeskeyfile

### Syntax

- For Zynq devices and FPGAs:

```
[aeskeyfile] <key filename>
```

- For Zynq UltraScale+ MPSoC:

```
[aeskeyfile = <keyfile name>] <partition>
```

- For Versal adaptive SoC:

```
{ aeskeyfile = <keyfile name>, file = <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 are 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:
{
    [keysrc_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:
{
    [keysrc_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_multitple.bif`

```
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.
- aeskeyfile format:

An `.nky` file accepts the following fields.

- **Device:** The name of the device for which the nky file is being used. Valid for both Zynq device and Zynq UltraScale+ MPSoC.
- **Keyx, IVx:** Here 'x' refers to an integer, that corresponds to the Key/IV number, for example, Key0, Key1, Key2 ..., IV0, IV1, IV2... An AES key must be 256 bits long while an IV key must be 12 bytes long. Keyx is valid for both Zynq devices and Zynq UltraScale+ MPSoC but IVx is valid only for Zynq UltraScale+ MPSoC.
- **Key Opt:** An optional key that you want to use to encrypt the first block of boot loader. Valid only for Zynq UltraScale+ MPSoC.
- **StartCBC - CBC Key:** An CBC key must be 128 bits long. Valid for Zynq devices only.
- **HMAC - HMAC Key:** An HMAC key must be 128 bits long. Valid for Zynq devices only.
- **Seed:** An initial seed that is used to generate the Key/IV pairs and needed to encrypt a partition. An AES Seed must be 256 bits long. Valid only for Zynq UltraScale+ MPSoC.
- **FixedInputData:** The data that is used as input to Counter Mode KDF, along with the Seed. An AES Fixed Input Data must be 60 Bytes long. Valid only for Zynq UltraScale+ MPSoC.

**Note:**

- Seed must be specified along with FixedInputData.
- Seed is not expected with multiple key/iv pairs.

## Versal Adaptive SoC Example

```
all:
{
  image
  {
    name = pmc_subsys, id = 0x1c000001
    {
      type = bootloader, encryption = aes,
      keysrc = bbam_red_key, aeskeyfile = key1.nky,
      file = plm.elf
    }
    {
      type = pmcdata, load = 0xf2000000,
      aeskeyfile = key2.nky, file = pmc_cdo.bin
    }
  }
  {
    type=cdo, encryption = aes,
```

```

        keysrc = efuse_red_key, aeskeyfile = key3.nky,
        file=fpd_data.cdo
    }
}

```

# alignment

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
{ alignment=<value>, file=<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

- For Zynq devices and Zynq UltraScale+ MPSoC:

```

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

```

- For Versal adaptive SoC:

```

new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core = a72-0, alignment=64, file = apu.elf }
    }
}

```

**Note:** \*base.pdi is the PDI generated by Vivado.

# 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.

**Note:** While there are different SPKs for the header and the FSBL, they share the same SPK ID.

If just the **auth\_params** field is used wherein the SPK ID is provided, the SPK ID propagates to the boot and application partitions. If SPK ID is used in both the boot and application partitions, the SPK ID in the boot/image header partition gets overwritten and application SPK is used. This means Bootgen chooses the last version of the SPK ID that is fed to it in the process of making sure that the header and FSBL have the same SPK ID.

- **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

```
all:  
{  
    [auth_params]ppk_select=0;spk_id=0x4  
    [pskfile] primary.pem  
    [sskfile]secondary.pem  
    [bootloader, authentication=rsa]fsbl.elf  
}
```

## Sample BIF 2 - test.bif

```
all:
{
    [auth_params] ppk_select=0;spk_select=spk-efuse;spk_id=0x22
    [pskfile]      primary.pem
    [sskfile]      secondary.pem
    [bootloader, authentication = rsa]
    fsbl.elf
}
```

## Sample BIF 3 - test.bif

```
all:
{
    [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

```
all:
{
    [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             = 0x3,
        spk_select         = spk-efuse,
        sskfile            = secondary1.pem
    ] fsbla53.elf

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

# authentication

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
{authentication=<options>, file=<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.
- ecdsa-p384 : Partition authenticated using ECDSA p384 curve
- ecdsa-p521 : Partition authenticated using ECDSA p521 curve

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    boot_config {bh_auth_enable}

    metaheader
    {
        authentication = rsa,
        pskfile = PSK2.pem,
        sskfile = SSK2.pem
    }

    image
```

```

{
    name = pmc_subsys, id = 0x1c000001
    partition
    {
        id = 0x01, type = bootloader,
        authentication = rsa,
        pskeyfile = PSK1.pem,
        sskfile = SSK1.pem,
        file = plm.elf
    }
    partition
    {
        id = 0x09, type = pmcdata, load = 0xf2000000,
        file = pmc_data.cdo
    }
}

image
{
    name = lpd, id = 0x4210002
    partition
    {
        id = 0x0C, type = cdo,
        authentication = rsa,
        pskeyfile = PSK3.pem,
        sskfile = SSK3.pem,
        file = lpd_data.cdo
    }
    partition
    {
        id = 0x0B, core = psm,
        authentication = rsa,
        pskeyfile = PSK1.pem,
        sskfile = SSK1.pem,
        file = psm_fw.elf
    }
}

image
{
    name = fpd, id = 0x420c003
    partition
    {
        id = 0x08, type = cdo,
        authentication = rsa,
        pskeyfile = PSK3.pem,
        sskfile = SSK3.pem,
        file = fpd_data.cdo
    }
}
}

```

# big\_endian

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
{ big_endian, file=<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

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
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
}
```

- For Versal adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core = a72-0, big_endian, file = apu.bin }
    }
}
```

Note: `*base.pdi` is the PDI generated by Vivado

---

## bbram\_kek\_iv

### Syntax

```
bbram_kek_iv = <iv file path>
```

### Description

This attribute specifies the IV that is used to encrypt the bbram black key. `bbram_kek_iv` is valid with `keysrc=bbram_blk_key`.

### Example

See [AES Encryption with Multiple Key Sources Example](#) for examples.

---

## bh\_kek\_iv

### Syntax

```
bh_kek_iv = <iv file path>
```

### Description

This attribute specifies the IV that is used to encrypt the boot header black key. `bh_kek_iv` is valid with `keysrc=bh_blk_key`.

### Example

See [AES Encryption with Multiple Key Sources Example](#) for examples.

---

## bh\_keyfile

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
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 obfuscated key or black key.

**Note:** Obfuscated key not supported for Versal devices.

## Arguments

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

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
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
}
```

- For Versal adaptive SoC:

```
all:
{
    bh_keyfile = bh_key1.txt
    bh_kek_iv = blk_iv.txt
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootloader, encryption = aes,
            keysrc = bbam_red_key, aeskeyfile = key1.nky, file = plm.elf
        }
        {
            type = pmcdata, load = 0xf2000000,
            aeskeyfile = key2.nky, file = pmc_cdo.bin
        }
        {
            type=cdo, encryption = aes,
            keysrc = bh_blk_key, aeskeyfile = key3.nky,
            file=fpd_data.cdo
        }
    }
}
```

---

## bh\_key\_iv

### Syntax

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

### Description

Initialization vector used when decrypting the black key.

### Arguments

Path to file.

### Example

```
Sample BIF - test.bif
all:
{
    [keysrc_encryption] bh_blk_key
    [bh_keyfile] bh_black_key.txt
    [bh_key_iv] bh_black_iv.txt
    [bootloader, encryption=aes, aeskeyfile=encl.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**

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
{ blocks = <size><num>;...;<size><*>, file=<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**

- <size>:** Specifies the size of blocks (in bytes).

**Example**

- For AMD Zynq™ UltraScale+™ MPSoC:

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

- For Versal adaptive SoC:

```
all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    metaheader
    {
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = efuse_red_metaheader_key.nky,
        dpacm_enable
    }

    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition
        {
            id = 0x01, type = bootloader,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = bbam_red_key.nky,
            dpacm_enable,
            blocks = 4096(2);1024;2048(2);4096(*),
            file = plm.elf
        }
        partition
        {
            id = 0x09, type = pmcdata, load = 0xf2000000,
            aeskeyfile = pmcdata.nky,
            file = pmc_data.cdo
        }
    }

    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0C, type = cdo,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = key1.nky,
            dpacm_enable,
            blocks = 8192(20);4096(*),
            file = lpd_data.cdo
        }
        partition
        {
            id = 0x0B, core = psm,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = key2.nky,
            dpacm_enable,
            blocks = 4096(2);1024;2048(2);4096(*),
            file = psm_fw.elf
        }
    }

    image
```

```

{
    name = fpd, id = 0x420c003
    partition
    {
        id = 0x08, type = cdo,
        encryption = aes,
        keysrc = bbram_red_key,
        aeskeyfile = key5.nky,
        dpacm_enable,
        blocks = 8192(20);4096(*),
        file = fpd_data.cdo
    }
}

```

**Note:** In the above example, the first two blocks are of 4096 bytes, followed by a block of 1024 bytes, and then the next two blocks are of 2048 bytes. The rest of the blocks are of 4096 bytes.

## boot\_config

### Syntax

```
boot_config { <options> }
```

### Description

This attribute specifies the parameters that are used to configure the bootimage. The options are:

- `bh_auth_enable`: Boot Header authentication enable, authenticating the bootimage while excluding the verification of PPK hash and SPK ID.
- `pufhd_bh`: PUF helper data is stored in boot header (default is efuse). PUF helper data file is passed to Bootgen using the option `puf_file`.
- `puf4kmode`: PUF is tuned to use in 4 k bit syndrome configuration (Default is 12 k bit).
- `shutter = <value>`: 32-bit PUF\_SHUT register value to configure PUF for shutter offset time and shutter open time.
- `smap_width = <value>`: Defines the SelectMAP (SMAP) bus width.

Options are:

- 8,16, 32 for monolithic/Master SLR (with default 32-bit)
- 0 for SSI technology Slave SLRs only

**Note:** SSI technology Slave SLRs will be set to `smap_width=0` to indicate the internal downstream connection. This option value must not be changed and is only applicable for SSI technology Slave SLRs.

- `dpacm_enable`: DPA Counter Measure Enable

- **a\_hwrot**: Asymmetric hardware root of trust (A-HWRoT) boot mode. Bootgen checks against the design rules for A-HWRoT boot mode. Valid only for production PDIs.
- **s\_hwrot**: Asymmetric hardware root of trust (S-HWRoT) boot mode. Bootgen checks against the design rules for S-HWRoT boot mode. Valid only for production PDIs.

### Examples

```
example_1:
{
    boot_config {bh_auth_enable, smap_width=16 }
    pskfile = primary0.pem
    sskfile = secondary0.pem
    image
    {
        {type=bootloader, authentication=rsa, file=plm.elf}
        {type=pmcdata, load=0xf2000000, file=pmc_cdo.bin}
    }
}
```

---

## boot\_device

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[boot_device] <options>
```

- For AMD Versal™ adaptive SoC:

```
boot_device { <options>, address=<address> }
```

### Description

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

### Arguments

Options for Zynq devices and Zynq UltraScale+ MPSoC:

- qspi32
- qspi24
- nand
- sd0
- sd1
- sd-ls

- mmc
- usb
- ethernet
- pcie
- sata

The address field specifies the offset of the image in the given flash device. Options for Versal adaptive SoC:

- qspi32
- qspi24
- nand
- sd0
- sd1
- sd-ls (SD0 (3.0) or SD1 (3.0))
- mmc
- usb
- ethernet
- pcie
- sata
- ospi
- smap
- sbi
- sd0-raw
- sd1-raw
- sd-ls-raw
- mmc1-raw
- mmc0
- mmc0-raw
- imagestore

**Example**

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    boot_device { qspi32, address=0x10000 }
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { id = 0x01, type = bootloader, file = plm.elf }
        { id = 0x09, type = pmcdata, load = 0xf2000000, file =
pmc_data.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        { id = 0x0C, type = cdo, file = lpd_data.cdo }
        { id = 0x0B, core = psm, file = psm_fw.elf }
    }
    image
    {
        name = pl_cfi, id = 0x18700000
        { id = 0x03, type = cdo, file = system.rcdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = fpd_data.cdo }
    }
}
```

## bootimage

**Syntax**

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[bootimage] <partition>
```

- For AMD Versal™ adaptive SoC:

```
{ type=bootimage, file=<partition> }
```



## Description

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

## Arguments

Specified file name.

## Example

- For FSBL:

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

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

- For `fsbl.bin` generation:

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

Use the following command:

```
bootgen -image fsbl.bif -o fsbl.bin -encrypt efuse
```

- For `system.bin` generation:

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

Use the following command:

```
bootgen -image system.bif -o system.bin
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
```

```
        name = apu_ss, id = 0x1c000000
        { load = 0x1000, file = system.dtb }
        { exception_level = el-2, file = u-boot.elf }
        { core = a72-0, exception_level = el-3, trustzone, file =
b131.elf }
    }
}
```

**Note:** `*base.pdi` is the PDI generated by Vivado.

---

## bootloader

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ type=bootloader, file=<partition> }
```

### Description

Identifies an ELF file as the FSBL or the PLM.

- 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

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
all:
{
    [bootloader] fsbl.elf
    hello.elf
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { id = 0x01, type = bootloader, file = plm.elf }
        { id = 0x09, type = pmcdata, load = 0xf2000000, file =
pmc_data.cdo }
    }
}
```

## bootvectors

### Syntax

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

### Description

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

### Example

```
all:
{
    [bootvectors]0x14000000,0x14000000,0x14000000,0x14000000,0x14000000,0x140000
00,0x14000000,0x14000000
    [bootloader,destination_cpu=a53-0]fsbl.elf
}
```

## checksum

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ checksum = <options>, file=<partition> }
```

## Description

This specifies the partition that 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 AMD Zynq™ 7000 SoC devices. In these devices, checksum operations are not supported for bootloaders.
- SHA3: Checksum operation for AMD Zynq™ UltraScale+™ MPSoC devices and Versal adaptive SoC.

## Examples

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
all:
{
    [bootloader] fsbl.elf
    [checksum=md5] hello.elf
}
```

- For AMD Versal™ adaptive SoC:

```
all:
{
    image
    {
        name = image1, id = 0x1c000001
        { type=bootloader, checksum=sha3, file=plm.elf }
        { type=pmcdata, file=pmc_cdo.bin }
    }
}
```

---

# copy

## Syntax

```
{ copy = <addr> }
```

## Description

This attribute specifies that the image is to be copied to memory at specified address.

### Example

```
test:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name=subsys_1, id=0x1c000000, copy = 0x30000
        { core=psm, file=psm.elf }
        { type=cdo, file=ps_data.cdo }
        { core=a72-0, file=a72_app.elf }
    }
}
```

---

## core

### Syntax

```
{ core = <options> }
```

### Description

This attributes specifies which core executes the partition.

### Arguments

- a72-0
- a72-1
- r5-0
- r5-1
- psm
- aie
- r5-lockstep

### Example

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
```

```
{
    name = apu_ss, id = 0x1c000000
    { core = a72-0, file = apu.elf }
}
```

**Note:** `*base.pdi` is the PDI generated by Vivado.

---

## delay\_auth

### Syntax

```
{delay_auth, file = filename}
```

### Description

This attribute indicates that the authentication is done at a later stage. This helps Bootgen to reserve space for hashes during partition encryption.

### Example

```
stage2b:
{
    image
    {
        name = lpd
        id = 0x4210002
        partition
        {
            id = 0x0C,
            type = cdo,
            encryption=aes, delay_auth
            keysrc = bbam_red_key,
            aeskeyfile = lpd_data.nky,
            file = lpd_data.cdo
        }
    }
}
```

---

## delay\_handoff

### Syntax

```
{ delay_handoff }
```

## Description

This attribute specifies that the hand-off to the subsystem is delayed.

## Example

```
test:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name=subsys_1, id=0x1c000000, delay_handoff
        { core=psm, file=psm.elf }
        { type=cdo, file=ps_data.cdo }
        { core=a72-0, file=a72_app.elf }
    }
}
```

---

# delay\_load

## Syntax

```
{ delay_load }
```

## Description

This attribute specifies that the loading of subsystem is delayed.

## Example

```
test:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name=subsys_1, id=0x1c000000, delay_load
        { core=psm, file=psm.elf }
        { type=cdo, file=ps_data.cdo }
        { core=a72-0, file=a72_app.elf }
    }
}
```

# destination\_cpu

## Syntax

```
[destination_cpu <options>] <partition>
```

## Description

Specifies which core executes the partition. The following example specifies that FSBL is 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 that the 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

```
[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

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

---

## early\_handoff

### Syntax

```
[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.

**Example**

```
all:
{
    [bootloader, destination_cpu=a53_0]fsbl.el
    [destination_cpu=r5-0]app1.elf
    [destination_cpu=r5-1,early_handoff]app2.elf
}
```

---

## efuse\_kek\_iv

**Syntax**

```
efuse_kek_iv = <iv file path>
```

**Description**

This attribute specifies the IV that is used to encrypt the efuse black key. So, 'efuse\_kek\_iv' is valid with 'keysrc=efuse\_blk\_key'.

**Example**

See [AES Encryption with Multiple Key Sources Example](#) for examples.

---

## efuse\_user\_kek0\_iv

**Syntax**

```
efuse_user_kek0_iv = <iv file path>
```

**Description**

This attribute specifies the IV that is used to encrypt the efuse user black key0. So, 'efuse\_user\_kek0\_iv' is valid with 'keysrc=efuse\_user\_blk\_key0'.

**Example**

See [AES Encryption with Multiple Key Sources Example](#) for examples.

---

## efuse\_user\_kek1\_iv

### Syntax

```
efuse_user_kek1_iv = <iv file path>
```

### Description

This attribute specifies the IV that is used to encrypt the efuse user black key1. So, 'efuse\_user\_kek1\_iv' is valid with 'keysrc=efuse\_user\_blk\_key1'.

### Example

See [AES Encryption with Multiple Key Sources Example](#) for examples.

---

## encryption

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ encryption = <options>, file = <filename> }
```

### 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**

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    metaheader
    {
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = efuse_red_metaheader_key.nky,
    }

    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition
        {
            id = 0x01, type = bootloader,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = bbam_red_key.nky,
            file = plm.elf
        }
        partition
        {
            id = 0x09, type = pmcdata, load = 0xf2000000,
            aeskeyfile = pmcdata.nky,
            file = pmc_data.cdo
        }
    }

    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0C, type = cdo,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = key1.nky,
            file = lpd_data.cdo
        }
        partition
        {
            id = 0x0B, core = psm,
            encryption = aes,
            keysrc = bbam_red_key,
```

```
        aeskeyfile = key2.nky,  
        file = psm_fw.elf  
    }  
}  
  
image  
{  
    name = fpd, id = 0x420c003  
    partition  
    {  
        id = 0x08, type = cdo,  
        encryption = aes,  
        keysrc = bbram_red_key,  
        aeskeyfile = key5.nky,  
        file = fpd_data.cdo  
    }  
}  
}
```

---

## exception\_level

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ exception_level=<options>, file=<partition> }
```

### Description

Exception level for which the core must be configured.

### Arguments

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

**Example**

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
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
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { load = 0x1000, file = system.dtb }
        { exception_level = el-2, file = u-boot.elf }
        { core = a72-0, exception_level = el-3,
trustzone, file = bl31.elf }
    }
}
```

**Note:** \*base.pdi is the PDI generated by Vivado.

---

## familykey

**Syntax**

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

**Description**

Specify family key. To obtain family key, contact an AMD representative at [secure.solutions@xilinx.com](mailto:secure.solutions@xilinx.com).

**Arguments**

Path to file.

### Example

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

---

## file

### Syntax

```
{ file = <path/to/file> }
```

### Description

This attribute specifies the file for creating the partition.

### Example

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core = a72-0, file = apu.elf }
    }
}
```

**Note:** `*base.pdi` is the PDI generated by Vivado.

---

## 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 is done excluding the verification of PPK hash and SPK ID.
- `auth_only`: Boot image is only RSA signed. FSBL should not be decrypted. See the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) for more information.
- `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 4 k bit configuration (default is 12 k bit).
- `shutter = <value>`: 32-bit PUF\_SHUT register value to configure PUF for shutter offset time and shutter open time.

**Note:** This shutter value must match the shutter value that was used during PUF registration.

## Example

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

---

# headersignature

## Syntax

For Zynq UltraScale+ MPSoC:

```
[headersignature] <signature file>
```

For Versal adaptive SoC:

```
headersignature = <signature file>
```

## Description

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



## Arguments

<signature\_file>

## Example

For Zynq UltraScale+ MPSoC:

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

For Versal adaptive SoC:

```
stage5:
{
    bhsignature = bootheader.sha384.sig

    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootimage,
            authentication=rsa,
            ppkfile = rsa-keys/PSK1.pub,
            spkfile = rsa-keys/SSK1.pub,
            spksignature = SSK1.pub.sha384.sig,
            file = pmc_subsys_e.bin
        }
    }
}
```

# hivec

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ hivec, file=<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

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    image
    {
        name = image1, id = 0x1c000001
        { type=bootloader, file=plm.elf }
        { type=pmcdata, file=pmc_cdo.bin }
        { type=cdo, file=fpd_data.cdo }
        { core=psm, file=psm.elf }
        { core=r5-0, hivec, file=hello.elf }
    }
}
```

# id

## Syntax

```
id = <id>
```

## Description

This attribute specifies the following IDs based on the place it is defined:

- `pdi` ID - within the outermost/PDI parenthesis
- `image` ID - within the image parenthesis

- partition ID - within the partition parenthesis

Image IDs are fixed for a given image. Refer to the following table for the image IDs defined by AMD for Versal adaptive SoCs.

**Table 38: Image IDs (Fixed for a Given Partition)**

Partition	Subsystem/Domain	Image ID Value	Description
PMC	Subsystem	0x1C000001	PMC subsystem ID
PLD	Domain	0x18700000	PLD0 Device ID (because PLD0 represents the entire PLD domain)
LPD	Domain	0x04210002	LPD Power Node ID
FPD	Domain	0x0420C003	FPD Power Node ID
Default Subsystem	Subsystem	0x1C000000	Default Subsystem ID
CPD	Domain	0x04218007	CPM Power Node ID
AIE	Domain	0x0421C005	AIE Power Node ID

**Note:** For AI Engine partitions and PS partitions, such as A72 and R5 ELF, use the default subsystem ID.

**Note:** Partition IDs are used for identifying a partition and are not used by PLM processing. The Partition IDs can be changed by the user according to their own numbering scheme. The PDI IDs and image IDs should not be changed.

### Example

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2                                // PDI ID
    image
    {
        name = pmc_subsys,
        id = 0x1c000001                    // Image ID
        partition
        {
            id = 0x01,                    // Partition ID
            type = bootloader,
            file = plm.elf
        }
    }
    {
        id = 0x09,
        type = pmcdata,
        load = 0xf2000000,
        file = pmc_data.cdo
    }
}
}
```

---

# image

## Syntax

```
image
{
}
```

## Description

This attribute is used to define a subsystem/image.

## Example

```
test:
{
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { type = bootloader, file = plm.elf }
        { type=pmcdata, load=0xf2000000, file=pmc_cdo.bin}
    }
    image
    {
        name = PL_SS, id = 0x18700000
        { id = 0x3, type = cdo, file = bitstream.rcdo }
        { id = 0x4, file = bitstream.rnpi }
    }
}
```

---

# imagestore

## Syntax

```
imagestore = <id>
```

## Description

To specify the ID of the PDI to add to Image Store. The Image Store feature allows PDI files to be stored in memory (DDR) and later be used to load specified images within the PDI. This is intended to allow partial reconfiguration, subsystem restart, and so on, without depending on external boot devices.

**Example**

```

write_imagestore_pdi:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0xb
    image
    {
        name = pl_noc, id = 0x18700000
        partition
        {
            id = 0xb05, type = cdo, file = imagestore.rnpi
        }
    }
}
master:
{
    id_code = 0x04d14093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = IMAGE_STORE, id = 0x18700000
        partition
        {
            id = 0xb15, imagestore = 0x1
            section = write_imagestore_pdi
        }
    }
}

```

## init

**Syntax**

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
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:

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    init = reginit.int
    image
    {
        name = image1, id = 0x1c000001
        { type=bootloader, file=plm.elf }
        { type=pmcdata, file=pmc_cdo.bin }
    }
}
```

---

## keysrc

### Syntax

```
keysrc = <options>
```

### Description

This specifies the Key source for encryption.

### Arguments

The valid key sources for boot loader, meta header and partitions are:

- efuse\_red\_key
- efuse\_blk\_key
- bbram\_red\_key
- bbram\_blk\_key
- bh\_blk\_key

There are few more key sources which are valid for partitions only:

- user\_key0
- user\_key1
- user\_key2
- user\_key3

- user\_key4
- user\_key5
- user\_key6
- user\_key7
- efuse\_user\_key0
- efuse\_user\_blk\_key0

### Example

```
all:
{
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootloader, encryption = aes,
            keysrc = bbram_red_key, aeskeyfile = key1.nky,
            file = plm.elf
        }
        {
            type = pmcdata, load = 0xf2000000,
            aeskeyfile = key2.nky, file = pmc_cdo.bin
        }
    }
}
```

---

## 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

```
all:
{
    [keysrc-encryption]efuse_gry_key
    [bootloader,encryption=aes, aeskeyfile=encl.nky,
destination_cpu=a53-0]fsbl.elf
}
```

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

## load

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ load = <value> , file=<partition> }
```

### Description

Sets the load address for the partition in memory.

### Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
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
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
}
```



```

image
{
    name = apu_ss, id = 0x1c000000
    { load = 0x1000, file = system.dtb }
    { exception_level = el-2, file = u-boot.elf }
    { core = a72-0, exception_level = el-3,
trustzone, file = bl31.elf }
    }
}

```

**Note:** `*base.pdi` is the PDI generated by Vivado.

## metaheader

### Syntax

```
metaheader { }
```

### Description

**Note:** All the security attributes are supported for `metaheader`.

This attribute is used to define encryption, authentication attributes for metaheaders such as keys, key sources, and so on.

### Example

```

test:
{
    metaheader
    {
        encryption = aes,
        keysrc = bbam_red_key,
        aeskeyfile = headerkey.nky,
        authentication = rsa
    }
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootloader,
            encryption = aes,
            keysrc = bbam_red_key,
            aeskeyfile = key1.nky,
            blocks = 8192(*),
            file = plm.elf
        }
        {
            type=pmcdata,
            load=0xf2000000,

```

```

        aeskeyfile=key2.nky,
        file=pmc_cdo.bin
    }
}

```

## name

### Syntax

```
name = <name>
```

### Description

This attribute specifies the name of the image/subsystem.

### Example

```

new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        { id = 0x01, type = bootloader, file = plm.elf }
        { id = 0x09, type = pmcdata, load = 0xf2000000, file =
pmc_data.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        { id = 0x0C, type = cdo, file = lpd_data.cdo }
        { id = 0x0B, core = psm, file = psm_fw.elf }
    }
    image
    {
        name = pl_cfi, id = 0x18700000
        { id = 0x03, type = cdo, file = system.rcdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = fpd_data.cdo }
    }
}

```

# offset

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ offset = <value>, file=<filename> }
```

## Description

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

## Arguments

Specified value and partition.

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
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
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { offset = 0x8000, file = data.bin }
    }
}
```

**Note:** \*base.pdi is the PDI generated by Vivado.

---

# overlay\_cdo

## Syntax

```
bootgen -arch versal -image test.bif -o test.bin -overlay_cdo ovl.cdo
```

## Description

The input file used with `overlay_cdo` command would have markers and content that needs to be overlayed. Bootgen searches for similar markers in all the cdo files present inside BIF, and when the content is found in that, cdo is replaced with the content from overlay cdo.

---

# parent\_id

## Syntax

```
parent_id = <id>
```

## Description

This attribute specifies the ID for the parent PDI. This is used to identify the relationship between a partial PDI and its corresponding boot PDI.

## Example

```
new_bif:
{
    id = 0x22
    parent_id = 0x2

    image
    {
        name = apu_ss, id = 0x1c000000
        { load = 0x1000, file = system.dtb }
        { exception_level = el-2, file = u-boot.elf }
        { core = a72-0, exception_level = el-3, trustzone, file = bl31.elf }
    }
}
```

# partition

## Syntax

```
partition
{
}
```

## Description

This attribute is used to define a partition. It is an optional attribute to make the BIF short and readable.

## Example

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys, id = 0x1c000001
        partition
        {
            id = 0x01,
            type = bootloader,
            file = plm.elf
        }
        partition
        {
            id = 0x09,
            type = pmcdata,
            load = 0xf2000000,
            file = pmc_data.cdo
        }
    }
}
```

**Note:** The partition attribute is optional and the BIF file can be written without the attribute too.

The above BIF can be written without the partition attribute as follows:

```
new_bif:
{
    id_code = 0x04ca8093
    extended_id_code = 0x01
    id = 0x2

    image
    {
        name = pmc_subsys, id = 0x1c000001
    }
}
```

```
        { id = 0x01, type = bootloader, file = plm.elf }  
        { id = 0x09, type = pmcdata, load = 0xf2000000, file =  
pmc_data.cdo }  
    }  
}
```

---

## partition\_owner, owner

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ owner = <options>, file=<filename> }
```

### Description

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

### Arguments

- For Zynq devices and Zynq UltraScale+ MPSoC:
  - fsbl: FSBL loads this partition
  - uboot: U-Boot loads this partition
- For AMD Versal™ adaptive SoC:
  - plm: PLM loads this partition
  - non-plm: PLM ignores this partition and it is loaded in a alternative way

### Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    image
    {
        { type = bootimage, file =
base.pdi }
    }
    image
    {
        name = apu_subsys, id = 0x1c000003
        {
            id = 0x00000000,
            core = a72-0,
            owner = non-plm,
            file = /path/to/image.ub
        }
    }
}
```

---

## pid

### Syntax

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

### Description

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

**Note:** If PID is not specified, it is incremented on every partition in the standard flow. If PID is not specified in the HSM flow, it always remains zero because each partition is handled separately. This behavior causes a mismatch between the final images.

### Example

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

---

## pmufw\_image

### Syntax

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

## Description

The PMU Firmware image that is 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

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
ppkfile = <filename>
```

## Description

The 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 PPK when the PSK is mentioned.



**Example**

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
  boot_config {bh_auth_enable}
  image
  {
    name = pmc_ss, id = 0x1c000001
    { type=bootloader, authentication=rsa, file=plm.elf,
ppkfile=primary0.pub, pskfile=primary0.pem,
sskfile=secondary0.pem }
    { type = pmcdata, load = 0xf2000000, file=pmc_cdo.bin }
    { type=cdo, authentication=rsa, file=fpd_cdo.bin,
ppkfile=primary1.pub, pskfile = primary1.pem, sskfile =
secondary1.pem }
  }
}
```

# presign

**Syntax**

For Zynq 7000 and Zynq UltraScale+ MPSoC devices:

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

For Versal adaptive SoC:

```
presign = <signature file>
```

**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 the <signature\_file> is applied.

### Example

For Zynq 7000 and Zynq UltraScale+ MPSoC devices:

```
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

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
pskfile = <filename>
```

### Description

This 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 PPK when the PSK is mentioned.

### Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    boot_config {bh_auth_enable}
    image
    {
        name = pmc-ss, id = 0x1c000001
        { type=bootloader, authentication=rsa, file=plm.elf,
          pskfile=primary0.pem, sskfile=secondary0.pem }
        { type = pmcdata, load = 0xf2000000, file=pmc_cdo.bin }
        { type=cdo, authentication=rsa, file=fpd_cdo.bin,
          pskfile = primary1.pem, sskfile = secondary1.pem }
    }
}
```

## puf\_file

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
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

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
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
}
```

- For AMD Versal™ adaptive SoC:

```
all:
{
  boot_config {puf4kmode}
  puf_file = pufhd_file_4K.txt
  bh_kek_iv = bh_black_key-iv.txt
  image
  {
    name = pmc_subsys, id = 0x1c000001
    {
      type = bootloader, encryption = aes,
      keysrc = bh_black_key, aeskeyfile = key1.nky,
      file = plm.elf
    }
    {
      type = pmcdata, load = 0xf2000000,
      aeskeyfile = key2.nky, file = pmc_cdo.bin
    }
    {
      type=cdo, encryption = aes,
      keysrc = efuse_red_key, aeskeyfile = key3.nky,
      file=fpd_data.cdo
    }
  }
}
```

## reserve

### Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
{ reserve = <value>, file=<filename> }
```

### Description

This attribute reserves the memory for a particular partition. Even if the partition size is lesser than the reserved memory, the partition length is always the reserved size. If the partition size is greater than the reserved size, then the partition length is the actual size of the partition.

This attribute is useful in cases where the users want to update the partitions in a bootimage without changing the corresponding header.

### Arguments

Specified partition

### Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { reserve = 0x1000, file = data.bin }
    }
}
```

**Note:** \*base.pdi is the PDI generated by Vivado.

---

## 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 Zynq UltraScale+ MPSoC, and normal is supported for both Zynq 7000 and Zynq UltraScale+ MPSoC. 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 scheduled to be deprecated.

**Note:** Split slaveboot mode is not supported for Versal adaptive SoC.

---

# spkfile

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
spkfile = <filename>
```

## Description

The SPK is used to authenticate partitions in the boot image. For more information, see [Using Authentication](#).

## Arguments

Specified file name.

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    boot_config {bh_auth_enable}
    pskfile=primary0.pem,
    image
    {
        name = pmc_ss, id = 0x1c000001
        { type=bootloader, authentication=rsa, file=plm.elf,
        spkfile=secondary0.pub,
          sskfile=secondary0.pem }
        { type = pmcdata, load = 0xf2000000, file=pmc_cdo.bin }
        { type=cdo, authentication=rsa, file=fpd_cdo.bin}
        spkfile=secondary1.pub, sskfile = secondary1.pem }
    }
}
```

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

# spksignature

## Syntax

For Zynq and Zynq UltraScale+ MPSoC devices:

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

For Versal adaptive SoC:

```
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

For Zynq and Zynq UltraScale+ MPSoC devices:

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

For Versal adaptive SoC:

```
stage7c:
{
    image
    {
        id = 0x1c000000, name = fpd
        { type = bootimage,
          authentication=rsa,
          ppkfile = PSK3.pub,
          spkfile = SSK3.pub,
          spksignature = SSK3.pub.sha384.sig,
          presign = fpd_data.cdo.0.sha384.sig,
          file = fpd_e.bin
        }
    }
}
```

---

# spk\_select

## Syntax

```
[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 is 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 is used for that partition.

## Example

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

    [
        bootloader,
        authentication = rsa,
        spk_select = spk-efuse,
        spk_id = 0x5,
        sskfile = ssk2.pem
    ] zynqmp_fsbl.elf

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

    [
        destination_cpu = a53-0,
        authentication = rsa,
        spk_select = spk-efuse,
        spk_id = 0x6,
        sskfile = ssk4.pem
    ] application2.elf
}
```

# sskfile

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
sskfile = <filename>
```

## Description

The SSK is used to authenticate partitions in the boot image. For more information, see [Using Authentication](#).

## Arguments

Specified file name.

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    boot_config {bh_auth_enable}
    image
    {
        name = pmc_ss, id = 0x1c000001
        { type=bootloader, authentication=rsa, file=plm.elf,
pskfile=primary0.pem, sskfile=secondary0.pem }
        { type = pmcdata, load = 0xf2000000, file=pmc_cdo.bin }
        { type=cdo, authentication=rsa, file=fpd_cdo.bin, pskfile =
primary1.pem, sskfile = secondary1.pem }
    }
}
```

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

# startup

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[startup = <value>] <filename>
```

- For AMD Versal™ adaptive SoC:

```
{ startup = <value>, file = <filename> }
```

## Description

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

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core=a72-0, load=0x1000, startup = 0x1000, file = apu.bin }
    }
}
```

**Note:** \*base.pdi is the PDI generated by Vivado.

# trustzone

## Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[trustzone = <options> ] <filename>
```

- For AMD Versal™ adaptive SoC:

```
{ trustzone = <options>, file = <filename> }
```

## Description

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

- secure
- nonsecure (default)

## Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
all:
{
    [bootloader, destination_cpu=a53-0] fsbl.elf
    [exception_level=e1-3, trustzone = secure] bl31.elf
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { load = 0x1000, file = system.dtb }
        { exception_level = e1-2, file = u-boot.elf }
        { core = a72-0, exception_level = e1-3, trustzone, file =
bl31.elf }
    }
}
```

**Note:** \*base.pdi is the PDI generated by Vivado.

---

# type

## Syntax

```
{ type = <options> }
```

## Description

This attribute specifies the type of partition. The options are as follows.

- bootloader
- pmcdata
- cdo
- bootimage

## Example

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core = a72-0, file = apu.elf }
    }
}
```

**Note:** `*base.pdi` is the PDI generated by Vivado.

---

# 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 AMD SoCs:

- zynq: 76 bytes
- zynqmp: 40 bytes

### Arguments

Specified file name.

### Example

```
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

```
123456789abcdef85072696e636530300301440408706d616c6c6164000508
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
}
```

---

## userkeys

### Syntax

```
userkeys = <filename>
```

### File Format

```
user_key0 <userkey0 value>
user_key1 <userkey1 value>
user_key2 <userkey2 value>
user_key3 <userkey3 value>
user_key4 <userkey4 value>
user_key5 <userkey5 value>
user_key6 <userkey6 value>
user_key7 <userkey7 value>
```

### Description

The path to the user keyfile. The keyfile contains user keys used to encrypt the partitions. The size of user key can be 128 or 256 bits. The 128-bit key can be used only for run-time loaded partitions.

### Example

In the following example, FPD partition uses the key source as `user_key2`, so the `.nky` file for this partition must have the `user_key2` from the `userkeys` file as the `key0`. This `key0` from the `.nky` file is then used by Bootgen for encryption. The PLM uses the `user_key2` programmed by `pmc_data` during decryption.

```
new_bif:
{
    userkeys = userkeyfile.txt
    id_code = 0x14ca8093
    extended_id_code = 0x01
    id = 0x2
    image
    {
        name = pmc_subsys
    }
}
```

```
id = 0x1c000001
partition
{
    id = 0x01
    type = bootloader
    encryption = aes
    keysrc=bbram_red_key
    aeskeyfile = inputs/keys/enc/bbram_red_key.nky
    dpacm_enable
    file = gen_files/plm.elf
}
partition
{
    id = 0x09
    type = pmcdata, load = 0xf2000000
    file = gen_files/pmc_data.cdo
}
}
image
{
    name = lpd
    id = 0x4210002
    partition
    {
        id = 0x0C
        type = cdo
        file = gen_files/lpd_data.cdo
    }
    partition
    {
        id = 0x0B
        core = psm
        file = static_files/psm_fw.elf
    }
}
image
{
    name = pl_cfi
    id = 0x18700000
    partition
    {
        id = 0x03
        type = cdo
        file = design_1_wrapper.rcdo
    }
    partition
    {
        id = 0x05
        type = cdo
        file = design_1_wrapper.rnpi
    }
}
image
{
    name = fpd
    id = 0x420c003
    partition
    {
        id = 0x08
        type = cdo
        file = gen_files/fpd_data.cdo
        encryption = aes
        keysrc=user_key2
    }
}
```



```
    aeskeyfile = userkey2.nky
  }
}
image
{
  name = ss_apu
  id = 0x1c000000
  partition
  {
    id = 0x61
    core = a72-0
    file = ./wrk_a72_r5/perip_a72/Debug/perip_a72.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 an AMD Zynq™ UltraScale+™ MPSoC device.

```
all:
{
  [bootloader, xip_mode] fsbl.elf
  application.elf
}
```

# Command Reference

See [Commands and Descriptions](#) for the device families supported by each of these commands.

---

## arch

### Syntax

```
-arch [options]
```

### Description

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

### Arguments

- zynq: AMD Zynq™ 7000 device architecture. This is the default value of the family architecture for which the boot image needs to be created.
- zynqmp: AMD Zynq™ UltraScale+™ MPSoC device architecture.
- fpga: Image is targeted for other FPGA architectures.
- versal: This image is targeted for AMD Versal™ devices.

### Return Value

None

### Example

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

---

# authenticatedjtag

## Syntax

```
-authenticatedjtag [options] [filename]
```

## Description

Used to enable JTAG during secure boot.

## Arguments

- rsa
- ecdsa

## Example

```
bootgen -arch versal -image boot.bif -w -o boot.bin -authenticatedjtag rsa  
authJtag-rsa.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 `-bif_help` on the command line.

---

# dual\_ospi\_mode

## Syntax

```
bootgen -arch versal -dual_ospi_mode stacked <size>
```

### Description

Generates two output files for dual OSPI stacked configuration, size (in MB) of the flash needs to be mentioned (64, 128, or 256).

### Example

This example generates two output files for independently programming to both flashes in a OSPI 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. This is only supported for Versal adaptive SoC.

```
bootgen -arch versal -image test.bif -o -boot.bin -dual_ospi_mode stacked 64
```

### Arguments

- stacked, <size>

---

## dual\_qspi\_mode

### Syntax

```
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, 32, 64, 128, or 256).

### Examples

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

```
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.

```
bootgen -image test.bif -o -boot.bin -dual_qspi_mode stacked 64
```

### Arguments

- parallel
- stacked <size>

---

## dump

### Syntax

```
-dump [options]
```

### Description

This command dumps the contents of boot header in to a separate binary file while generating PDI.

### Example

```
[bootgen -image test.bif -o -boot.bin -log trace -dump bh]
```

### Arguments

- empty: Dumps the partitions as binary files.
- bh: Dumps boot header as a separate file.

**Note:** Boot header is dumped as a separate binary file along with PDI. PDI generated is not be stripped of the boot header, but it retains the boot header.

---

## dump\_dir

### Syntax

```
dump_dir <path>
```

### Description

This option is used to specify a directory location to write the contents of -dump command.

### Example

```
bootgen -arch versal -dump boot.bin -dump_dir <path>
```

---

# efuseppkbits

## Syntax

```
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:

- AMD Zynq™ 7000 uses the SHA2 protocol for hashing.
- AMD Zynq™ UltraScale+™ MPSoC and Versal adaptive SoC 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 AMD Zynq™ UltraScale+™ MPSoC.

## Example

```
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 AMD Zynq™ 7000 format:

*Table 39: Zynq: SHA-2 (256-bytes)*

Value	SHA-2 Hash	T-Padding	0x0	0xFF	0x01	0x00
Number of bytes	32	19	1	202	1	1

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

*Table 40: ZynqMP: SHA-3 (384-bytes)*

Value	0x0	0x1	0xFF	0xFF	T-Padding	SHA-3 Hash
Number of bytes	1	1	314	1	19	48

## 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

```
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:

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

The Sample BIF file is shown in the following example:

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

## Arguments

- rsa
- pem

- obfuscated

---

## h, help

### Syntax

```
bootgen -help  
bootgen -help arch
```

### Description

Lists the supported command line attributes. For a more detailed explanation of each attribute, specify the attribute name as argument to `-help` on the command line.

---

## 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:

```
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 information is captured.
- `trace`: More detailed information is captured along with the information above.

---

## nonbooting

### Syntax

```
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

```
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

```
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.

**Note:** MCS format bootimage/PDI generation is supported for AMD Zynq™/AMD Zynq™MP and AMD Versal™ monolithic devices.

## p

### Syntax

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

### Description

This option specifies the partname of the AMD 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 `xc7z020c1g484` 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 the 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

- For Zynq devices, the maximum partition is 14.
- For Zynq UltraScale+ MPSoCs, the maximum partition is 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] <filename>
```

### Description

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

### Arguments

- bh: To read boot header from boot image in human readable form
- iht: To read image header table from boot image
- ih: To read image headers from boot image.
- pht: To read partition headers from boot image
- ac: To read authentication certificates from boot image

### Example

```
bootgen -arch zynqmp -read BOOT.bin
```

**Note:** -read is not supported for bootimages/PDIs in MCS format

---

# spksignature

## Syntax

```
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

```
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

```
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 are verified against the available partitions. Verification is performed in the following steps:

1. Verify header authentication certificate:
  - For Zynq UltraScale+ MPSoC: verify SPK signature and verify header signature.
  - For Versal: verify SPK signature, verify IHT signature, and verify meta header signature.
2. Verify bootloader authentication certificate: verify boot header signature, verify SPK signature, and verify bootloader signature.
3. Verify partition authentication certificate: verify SPK signature and verify partition 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 = d54b6fd94f7cf98fd955517f937e9927f9536caebe148fba1818c1ba46bba3a4
FixedInputDataByteLen = 60
FixedInputData =
94c4a0c69526196c1377cebf0a2ae0fb4b57797c61bea8eeb0518ca08652d14a5e1bd1b116b1
794ac8a476acbdbbcd4f6142d7b8515bad09ec72f7af
```

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 is printed on the console for you 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 is 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 is 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 processing 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
ulo divide
+ = addition
- = subtraction
~ = negation
>> = shift right
<< = shift left
& = binary and
  = binary or
^ = binary nor
```

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.

# CDO Utility

The CDO utility (cdoutil) is a program that allows to process CDO files in various ways. CDO files are binary files created in the AMD Vivado™ Design Suite for AMD Versal™ devices based on user configuration for clocks, PLLs, and MIO. CDOs are part of the PDI, and are loaded/executed by the PLM. For AMD Zynq™ 7000 devices and AMD Zynq™ UltraScale+™ MPSoCs, the configuration is part of `ps7/pvu_init.c/h` files, which are compiled along with the FSBL.

## Accessing

The cdoutil is available as part of the Vivado Design Suite/AMD Vitis™ unified software platform/Bootgen installation at `<INSTALL_DIR>/bin/cdoutil`.

## Usage

The general command line syntax for cdoutil is:

```
cdoutil <options> <input(s)>
```

The default function of cdoutil is to decode the input file and print out the CDO.

## Command Line Options

There are a number of options to change the default behavior:

**Table 41: Command Line Options**

Option	Description
-address-filter-file <path>	Specify address filter file
-annotate	Annotate source output with details of commands
-device <type>	Specify device name, default is xcvc1902
-help	Print help information
-output-binary-be	Output CDO commands in big endian binary format
-output-binary-le	Output CDO commands in little endian binary format

Table 41: Command Line Options (cont'd)

Option	Description
<code>-output-file &lt;path&gt;</code>	Specify output file, default is stdout
<code>-output-modules</code>	Output list of modules used by input file(s)
<code>-output-raw-be</code>	Output CDO commands in big endian raw format
<code>-output-raw-le</code>	Output CDO commands in little endian raw format
<code>-output-source</code>	Output CDO commands in source format (default)
<code>-remove-comments</code>	Remove comments from input
<code>-rewrite-block</code>	Rewrite block write commands to multiple write commands
<code>-rewrite-sequential</code>	Rewrite sequential write commands to a single block write command
<code>-verbose</code>	Print log information
<code>post-process &lt;mode&gt;</code>	Post process PMCFW commands to PLM commands
<code>cfu-stream-keyhole-size &lt;size&gt;</code>	Override default CFU stream keyhole size
<code>random-commands &lt;count&gt;</code>	Generate <count> random commands
<code>apropos &lt;keywords&gt;</code>	Search device register information for <keywords> and output matches
<code>overlay &lt;path&gt;</code>	Specify overlay file

**Note:** `-output-raw-be` is preferred as the Vivado Design Suite produces CDOs in big endian raw format. `-output-raw-le`, `-output-binary-be`, and `-output-binary-le` are not preferred options.

## Address Filter File

The address filter file is specified using the `-address-filter-file <path>`. The purpose of this file is to specify modules that must be removed from the configuration. The address filter file is a text file where each line starting with the dash (minus) character specifies an address range for which all initializations should be removed. Example:

```
# Remove configuration of UART0
-UART0
```

The list of modules used in a design can be generated using the `-output-modules` option. This can be a useful starting point for the address filter file.

## Examples

### Converting Binary to Source without Annotations

```
cdoutil -output-file test.txt test.bin
```

Example output:

```
version 2.0
write 0xfca50000 0
write 0xfca50010 0
write 0xfca50018 0x1
write 0xfca5001c 0
write 0xfca50020 0
write 0xfca50024 0xffffffff
```

## Converting Binary to Source with Annotations

```
cdoutil -annotate -output-file test.txt test.bin
```

Example output:

```
version 2.0
# PCIEA_ATTRIB_0.MISC_CTRL.slvrr_enable[0]=0x0
write 0xfca50000 0
# PCIEA_ATTRIB_0.ISR.{dp11_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x0}
write 0xfca50010 0
# PCIEA_ATTRIB_0.IER.{dp11_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x1}
write 0xfca50018 0x1
# PCIEA_ATTRIB_0.IDR.{dp11_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x0}
write 0xfca5001c 0
# PCIEA_ATTRIB_0.ECO_0.eco_0[31:0]=0x0
write 0xfca50020 0
# PCIEA_ATTRIB_0.ECO_1.eco_1[31:0]=0xffffffff
write 0xfca50024 0xffffffff
```

## Editing Binary CDO File

```
cdoutil -annotate -output-file test.txt test.bin
vim test.txt
cdoutil -output-binary-be -output-file test-new.bin test.txt
```

Make sure .bif file is using test-new.bin instead of test.bin, then rerun bootgen to create the .pdi file.

## Converting Source to Binary

```
cdoutil -output-binary-be -output-file test.bin test.txt
```

# Design Advisories for Bootgen

- AMD recommends that you generate your own keys for fielded systems and then provide those keys to the development tools. See [AR#76171](#) for more information.
- In this release, few encryption key rolling blocks are supported for Versal. See [AR#76515](#) for more information.

# Additional Resources and Legal Notices

---

## Finding Additional Documentation

### Documentation Portal

The AMD Adaptive Computing Documentation Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Documentation Portal, go to <https://docs.xilinx.com>.

### Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

### Design Hubs

AMD 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.
- Go to the [Design Hubs](#) webpage.

**Note:** For more information on DocNav, see the [Documentation Navigator](#) webpage.

---

## Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Support](#).

---

## Additional Resources

### Bootgen Document Resources

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

- For AMD Zynq™ 7000 SoC:
  - *Zynq-7000 SoC Software Developers Guide* ([UG821](#))
  - *Zynq-7000 SoC Technical Reference Manual* ([UG585](#))
- For AMD Zynq™ UltraScale+™ MPSoC:
  - *Zynq UltraScale+ MPSoC: Software Developers Guide* ([UG1137](#))
  - *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#))
  - AMD Zynq UltraScale+ MPSoC Solution Center: <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841847/Solution+ZynqMP+PL+Programming>
- For AMD Versal™ adaptive SoC:
  - *Xilinx Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* ([UG1305](#))
  - *Versal Adaptive SoC System Software Developers Guide* ([UG1304](#))
  - *Versal Adaptive SoC Technical Reference Manual* ([AM011](#))

### Other Document Resources for Bootgen

- *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))
- *Zynq 7000 SoC Secure Boot Getting Started Guide* ([UG1025](#))
- [Xilinx Software Command-Line Tool](#) in the *Vitis Embedded Software Development Flow Documentation* ([UG1400](#))
- *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* ([UG1209](#))
- *Secure Boot of Zynq 7000 SoC* ([XAPP1175](#))
- *Run Time Integrity and Authentication Check of Zynq 7000 SoC System Memory* ([XAPP1225](#))
- *Programming BBRAM and eFUSES* ([XAPP1319](#))



- *Versal Adaptive SoC Security Manual (UG1508)*. This manual requires an active NDA to be downloaded from the [Design Security Lounge](#).

## Revision History

The following table shows the revision history for this document.

Section	Revision Summary
<b>06/15/2023 Version 2023.1</b>	
General updates	Editorial updates only. No technical content updates.
<b>05/16/2023 Version 2023.1</b>	
<a href="#">imagestore</a>	Added topic.
<b>01/02/2023 Version 2022.2</b>	
General updates	Editorial updates only. No technical content updates.
<b>12/23/2022 Version 2022.2</b>	
<a href="#">Chapter 6: SSIT Support</a>	General updates.
<b>12/14/2022 Version 2022.2</b>	
<a href="#">Chapter 6: SSIT Support</a>	Updated for Versal HBM.
<b>10/19/2022 Version 2022.2</b>	
<a href="#">Chapter 6: SSIT Support</a>	Added SSI technology Authentication flow.
N/A	Bug fixes.
<b>04/26/2022 Version 2022.1</b>	
<a href="#">Chapter 4: Using Bootgen GUI</a>	Updated Images
<a href="#">Chapter 6: SSIT Support</a>	Updated for Versal Premium
<a href="#">Chapter 9: BIF Attribute Reference</a>	Updated BIF Attributes
<a href="#">Chapter 10: Command Reference</a>	Updated Command References

## Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of

AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### **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 2019-2023 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, Versal, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. 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 US and/or elsewhere. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.