

 查看所有版本

Bootgen 用户指南

UG1283 (v2024.2) 2024 年 12 月 13 日

本文档为英语文档的翻译版本，若译文与英语原文存在歧义、差异、不一致或冲突，概以英语文档为准。译文可能并未反映最新英语版本的内容，故仅供参考，请参阅最新版本的英语文档获取最新信息。

AMD 自适应计算矢志不渝地为员工、客户与合作伙伴打造有归属感的包容性环境。为此，我们正从产品和相关宣传资料中删除非包容性语言。我们已发起内部倡议，以删除任何排斥性语言或者可能固化历史偏见的语言，包括我们的软件和 IP 中嵌入的术语。虽然在此期间，您仍可能在我们的旧产品中发现非包容性语言，但请确信，我们正致力于践行革新使命以期与不断演变的行业标准保持一致。如需了解更多信息，请参阅此[链接](#)。



目录

第 1 章：简介	6
安装 Bootgen	6
启动时间安全	6
第 2 章：启动镜像布局	8
Zynq 7000 SoC 启动和配置	8
Zynq UltraScale+ MPSoC 启动和配置	16
Versal 自适应 SoC 启动镜像格式	28
第 3 章：创建启动镜像	41
启动镜像格式 (BIF)	41
BIF 语法和受支持的文件类型	42
属性	45
第 4 章：使用 Bootgen GUI	53
启动 Bootgen GUI	53
适用于 Zynq 7000 器件和 Zynq UltraScale+ 器件的 Bootgen GUI	53
为 Versal 自适应 SoC 使用 Bootgen GUI 选项	55
在命令行上使用 Bootgen	57
命令和描述	57
第 5 章：启动时间安全	61
使用加密	62
使用身份验证	71
Versal 身份验证支持	80
Versal 散列方案	82
使用 HSM 模式	83
第 6 章：SSIT 支持	112
第 7 章：FPGA 支持	124
加密和身份验证	124
HSM 模式	125
同时使用身份验证和加密的 HSM 流程	127
附录 A：用例与示例	129
Zynq MPSoC 用例	129
Versal 自适应 SoC 用例	138

附录 B: BIF 属性参考.....	150
aarch32_mode.....	150
aeskeyfile.....	151
alignment.....	154
auth_params.....	154
authentication.....	156
big_endian.....	158
bbram_kek_iv.....	159
bh_kek_iv.....	159
bh_keyfile.....	160
bh_key_iv.....	161
bhsignature.....	161
blocks.....	162
boot_config.....	164
boot_device.....	165
bootimage.....	167
bootloader.....	168
bootvectors.....	169
checksum.....	170
copy.....	171
core.....	171
delay_auth.....	172
delay_handoff.....	173
delay_load.....	173
destination_cpu.....	174
destination_device.....	175
early_handoff.....	175
efuse_kek_iv.....	176
efuse_user_kek0_iv.....	176
efuse_user_kek1_iv.....	176
encryption.....	177
exception_level.....	179
familykey.....	180
file.....	180
fsbl_config.....	181
headersignature.....	181
hivec.....	182
id.....	183
image.....	184
imagestore.....	185
init.....	186
keysrc.....	187
keysrc_encryption.....	188
load.....	188
metaheader.....	189
name.....	190

offset.....	191
optionaldata.....	192
overlay_cdo.....	193
parent_id.....	193
partition.....	193
partition_owner 和 owner.....	194
pid.....	196
pmufw_image.....	196
ppkfile.....	197
presign.....	198
pskfile.....	198
puf_file.....	199
reserve.....	200
split.....	201
spkfile.....	202
spksignature.....	203
spk_select.....	204
sskfile.....	205
startup.....	206
trustzone.....	207
type.....	208
udf_bh.....	208
udf_data.....	209
userkeys.....	210
xip_mode.....	212
附录 C：命令参考.....	213
arch.....	213
authenticatedjtag.....	213
bif_help.....	214
dual_ospo_mode.....	214
dual_qspi_mode.....	215
dump.....	215
dump_dir.....	216
efuseppkbits.....	216
enable_auth_opt.....	217
encrypt.....	217
encryption_dump.....	218
fill.....	218
generate_hashes.....	219
generate_keys.....	220
h 和 help.....	221
image.....	221
log.....	222
nonbooting.....	222
o.....	223
p.....	223



padimageheader.....	223
process_bitstream.....	224
read.....	224
spksignature.....	225
split.....	225
verify.....	226
verify_kdf.....	226
w.....	227
zynqmpes1.....	227
初始化对和 INT 文件属性.....	228
附录 D：CDO 实用工具.....	229
访问.....	229
用法.....	229
示例.....	230
附录 E：Bootgen 设计咨询.....	232
附录 F：附加资源与法律声明.....	233
查找其他文档.....	233
支持资源.....	233
参考资料.....	234
修订历史.....	234
请阅读：重要法律声明.....	236

简介

AMD FPGA、系统级芯片 (SoC) 器件和自适应 SoC 通常包含多个软硬件二进制文件，这些二进制文件用于启动器件并使其按设计和期望的方式运行。这些二进制文件可包含可采用非安全方法和安全方法加载的 FPGA 比特流、固件镜像、启动加载程序、操作系统和用户所选的应用。

Bootgen 是一款 AMD 工具，它支持您将二进制文件缝合在一起，并生成器件启动镜像。Bootgen 定义了多个属性和参数作为创建启动镜像时的输入，以供 AMD 器件使用。

AMD 器件的安全启动功能使用的是公钥和私钥加密算法。Bootgen 可提供具体的目标存储器地址分配和对应每个分区的对齐要求。它还支持加密和身份验证，如 [使用加密](#) 和 [使用身份验证](#) 中所述。在 [使用 HSM 模式](#) 中描述了更高级的身份验证流程和密钥管理选项，其中 Bootgen 可输出中间散列文件，可使用私钥对这些中间散列文件进行脱机签名，以便对启动镜像中包含的身份验证证书进行签名。Bootgen 通过向分区列表添加报头块来汇编启动镜像。（可选）每个分区均可使用 Bootgen 来进行加密和身份验证。输出为单个文件，此文件可直接烧录到系统的启动闪存中。该工具还可生成多个输入文件以支持身份验证和加密。如需了解更多信息，请参阅 [BIF 语法和受支持的文件类型](#)。

Bootgen 随附有 GUI 界面和命令行选项。该工具集成到 AMD Vitis™ 集成设计环境 (IDE) 中，用于生成基本启动镜像，但大部分 Bootgen 选项均由命令行驱动。命令行选项可采用脚本编制。Bootgen 工具由启动镜像格式 (BIF) 配置文件驱动，文件扩展名为 *.bif。搭配 AMD SoC 和自适应 SoC 使用时，Bootgen 能够为 AMD 7 系列和更高版本的 FPGA 进行分区加密和身份验证，如 [第 7 章：FPGA 支持](#) 中所述。除支持用于定义启动镜像行为的命令和属性外，还有其他多个实用工具可帮助您使用 Bootgen。现已可在 [GitHub](#) 上获取 Bootgen 代码。

安装 Bootgen

您可在 Vitis IDE 的 GUI 模式下使用 Bootgen 来执行简单的启动镜像创建操作，或者在命令行模式下创建更复杂的启动镜像。您可使用 AMD 统一安装程序（适用于 FPGA 和自适应 SoC）安装 Bootgen。

安装含 Bootgen 的 Vitis 后，您即可通过以下任一方式启动并使用该工具：通过 Vitis IDE 选项（其中包含用于快速开发和实验的常用操作），或者通过软件命令行工具 (XSDB)。

命令行选项包含许多其他用于创建启动镜像的选项。如需了解更多信息，请参阅 [在命令行上使用 Bootgen 选项](#)。

启动时间安全

支持采用最新身份验证方法来实现安全启动，这样可防止在 AMD 器件上运行未经授权或经修改的代码，并确保仅限已授权的程序才能访问镜像以加载各种加密技术。

对于特定于器件的硬件安全性功能，请参阅以下文档：

- 《Zynq 7000 SoC 技术参考手册》 ([UG585](#))
- 《Zynq UltraScale+ 器件技术参考手册》 ([UG1085](#))

- 《Versal 自适应 SoC 技术参考手册》(AM011)

注释：如需了解其他信息，请参阅《Versal 自适应 SoC 安全手册》(UG1508)。本手册需要从[设计安全性专区](#)下载有效的 NDA。

如需了解有关使用 Bootgen 时对内容进行加密和身份验证的更多信息，请参阅[使用加密](#) 和 [使用身份验证](#)。

Bootgen 硬件安全模式 (HSM) 有助于提升密钥处理安全性，因为 BIF 属性使用公钥，而非 RSA 私钥。HSM 是安全密钥/签名生成器件，可生成私钥、使用该私钥对分区进行加密，并将 RSA 密钥的公用部分提供给 Bootgen。私钥始终保留在 HSM 内。Bootgen HSM 模式的 BIF 使用由 HSM 生成的公钥和签名。如需了解更多信息，请参阅[使用 HSM 模式](#)。

启动镜像布局

本节描述了不同架构的启动镜像的格式。

- 如需了解有关将 Bootgen 用于 Zynq 7000 器件的信息，请参阅 [Zynq 7000 SoC 启动和配置](#)
- 如需了解有关将 Bootgen 用于 AMD Zynq™ UltraScale+™ MPSoC 器件的信息，请参阅 [Zynq UltraScale+ MPSoC 启动和配置](#)
- 如需了解有关如何将 Bootgen 用于 AMD FPGA 的信息，请参阅 [第 7 章：FPGA 支持](#)
- 如需了解有关 AMD Versal™ 自适应 SoC 的信息，请参阅 [Versal 自适应 SoC 启动镜像格式](#)

构建启动镜像包含下列步骤：

1. 创建 BIF 文件。
2. 运行 Bootgen 可执行文件以创建启动镜像。

注释：对于 Quick Emulator (QEMU)，必须将二进制文件转换为对应于启动器件的镜像格式。

每个器件的输出文件可能相同（例如，对于每个器件，ELF 文件可以是包含在启动镜像内的输入文件），但启动镜像的格式不同。以下章节描述了每个器件的启动报头、镜像报头、分区报头、初始化和身份验证证书报头所需的格式。

Zynq 7000 SoC 启动和配置

本章描述了 Zynq 7000 SoC 器件的启动和配置顺序。如需了解有关可用的第一阶段启动加载程序 (FSBL) 结构的更多详细信息，请参阅《Zynq 7000 SoC 技术参考手册》([UG585](#))。

Zynq 7000 SoC 上的 BootROM

BootROM 是应用处理单元 (APU) 中运行的首个软件。BootROM 将在首个 Cortex® 处理器 A9-0 上执行，而第二个处理器 Cortex A9-1 则执行等待事件 (WFE) 指令。BootROM 的主要任务是配置系统、将 FSBL 从启动器件复制到片上存储器 (OCM)，然后将代码执行通过分支拆分到 OCM。

(可选) 在非安全环境内，可从 QSPI 或 NOR 器件直接执行 FSBL。主启动器件包含一个或多个启动镜像。每个启动镜像均由启动报头和第一阶段启动加载程序 (FSBL) 组成。此外，启动镜像可包含可编程逻辑 (PL)、第二阶段启动加载程序 (SSBL) 以及嵌入式操作系统和应用；但这些均不可供 BootROM 访问。BootROM 执行流程受启动模式管脚捆绑设置、启动报头以及发现的有关系统的信息的影响。BootROM 可在安全环境内以加密 FSBL 来执行，或者也可在非安全环境内执行。受支持的启动模式包括：

- JTAG 模式主要用于开发和调试。
- NAND、并行 NOR、串行 NOR (QSPI) 和安全数字 (SD) 闪存用于启动器件。

《Zynq 7000 SoC 技术参考手册》([UG585](#)) 提供这些启动模式的详细信息。如需获取有关常见启动和配置问题的答案，请参阅[答复记录 52538](#)。

Zynq 7000 SoC 启动镜像布局

以下是可包含在 AMD Zynq™ 7000 SoC 启动镜像内的组件图示。

图 1：启动报头



X25912-102521

Zynq 7000 SoC 启动报头

此外，启动报头还包含 [Zynq 7000 SoC 寄存器初始化表](#)。在向 FSBL 交接控制权之前，BootROM 使用启动报头来查找 FSBL 的位置和长度信息以及有关对系统进行初始化的详细信息。

Bootgen 将启动报头附加到某一启动镜像开头处。启动报头表为包含主启动加载程序（如 FSBL）相关启动信息的结构。此结构在整个启动镜像中唯一。该表由 BootROM 解析，以判定闪存中 FSBL 存储位置以及 OCM 中所需的 FSBL 加载位置。其中还存储有部分加密和身份验证相关的参数。

其他启动镜像组件包括：

- [Zynq 7000 SoC 镜像报头表](#)
- [Zynq 7000 SoC 镜像报头](#)
- [Zynq 7000 SoC 分区报头](#)
- [Zynq 7000 SoC 身份验证证书](#)

下表提供了 AMD Zynq™ 7000 SoC 启动报头的地址偏移、参数和描述。

表 1：Zynq 7000 SoC 启动报头

地址偏移	参数	描述
0x00-0x1F	Arm® Vector table	由 Bootgen 使用虚拟矢量表填充（Arm 操作代码 0xEFFFFFFE，即用于捕获未初始化矢量的 branch-to-self 无限循环）。
0x20	Width Detection Word	此项是识别单堆叠模式、双堆叠模式或双并行模式下的 QSPI 闪存所必需的。0xAA995566（小字节序格式）。
0x24	Header Signature	包含 4 字节的“X”、“N”、“L”、“X”（按字节顺序），按小字节序格式为 0x584c4e58。
0x28	Key Source	器件中加密密钥的位置： 0x3A5C3C5A：BBRAM 中的加密密钥。 0xA5C3C5A3：eFUSE 中的加密密钥。 0x00000000：未加密。
0x2C	Header Version	0x01010000
0x30	Source Offset	此镜像文件中 FSBL（启动加载程序）的位置。
0x34	FSBL Image Length	解密后 FSBL 的长度。
0x38	FSBL Load Address (RAM)	FSBL 要复制到的目标 RAM 地址。
0x3C	FSBL Execution address (RAM)	FSBL 执行的入口矢量。
0x40	Total FSBL Length	加密后 FSBL 的总大小，包括身份验证证书（如有）和填充。
0x44	QSPI Configuration Word	硬编码为 0x00000001。
0x48	Boot Header Checksum	按标准算法，从偏移 0x20 到 0x44（含）计算所得的字数总和的反码。这些字假定按小字节序。
0x4c-0x97	User Defined Fields	76 字节
0x98	Image Header Table Offset	指向镜像报头表的指针
0x9C	Partition Header Table Offset	指向分区报头表的指针

Zynq 7000 SoC 寄存器初始化表

Bootgen 中的“寄存器初始化表”采用 256 个地址/值对格式，用于为 MIO 多路复用器和闪存时钟初始化 PS 寄存器。欲知详情，请参阅 [关于寄存器初始化对和 INT 文件属性](#)。

表 2：Zynq 7000 SoC 寄存器初始化表

地址偏移	参数	描述
0xA0 到 0x89C	寄存器初始化对：<address>:<value>：	Address = 0xFFFFFFFF 表示跳过该寄存器并忽略该值。 所有未使用的寄存器字段都必须设置为 Address=0xFFFFFFFF 和 value = 0x0。

Zynq 7000 SoC 镜像报头表

Bootgen 通过从 ELF 文件、比特流、数据文件等中提取数据来创建启动镜像。从中提取数据的这些文件被称为镜像。每个镜像都包含一个或多个分区。“Image Header”（镜像报头）表采用包含所有镜像公用的信息以及如下信息的结构：镜像数量、启动镜像中存在的分区数量以及指向其他报头表的指针。下表提供了 AMD Zynq™ 7000 SoC 器件的地址偏移、参数和描述。

表 3：Zynq 7000 SoC 镜像报头表

地址偏移	参数	描述
0x00	Version	0x01010000：仅包含以下可用字段：0x0、0x4、0x8、0xC 和填充 0x01020000:0x10 字段为添加的字段。
0x04	Count of Image Headers	表示镜像报头的数量。
0x08	First Partition Header Offset	指向首个分区报头的指针。（字偏移）
0x0C	First Image Header Offset	指向首个镜像报头的指针。（字偏移）
0x10	Header Authentication Certificate Offset	指向身份验证证书报头的指针。（字偏移）
0x14	Reserved	默认为 0xFFFFFFFF。

Zynq 7000 SoC 镜像报头

“Image Header”（镜像报头）为阵列结构，其中包含每个镜像的相关信息，例如，ELF 文件、比特流、数据文件等。每个镜像都可具有多个分区，例如，每个 ELF 均可包含多个可加载节，每个节均构成启动镜像中的一个分区。该表还包含镜像相关分区数量的信息。下表提供了 AMD Zynq™ 7000 SoC 器件的地址偏移、参数和描述。

表 4：Zynq 7000 SoC 镜像报头

地址偏移	参数	描述
0x00	Next Image Header	链接到下一个镜像报头。如果当前镜像报头为最后一个镜像报头，则为 0（字偏移）。
0x04	Corresponding partition header	链接到首个关联的分区报头（字偏移）。
0x08	Reserved	始终为 0。
0x0C	Partition Count Length	与此镜像关联的分区数量。
0x10 到 N	Image Name	以大字节序打包。为对该字符串进行重构，请每次解包 4 字节、反转顺序然后串联。例如，字符串 “FSBL10.ELF” 打包为 0x10：‘L’，‘B’，‘S’，‘F’，0x14：‘E’，‘.’，‘；’，‘0’，‘1’；0x18：‘\0’，‘\0’，‘P’，‘I’。打包的镜像名称为 4 字节的倍数。
N	String Terminator	0x00000000
N+4	Reserved	默认为 0xFFFFFFFF，边界为 64 字节。

Zynq 7000 SoC 分区报头

“Partition Header”（分区报头）为阵列结构，其中包含每个分区的相关信息。每个分区报头表均由启动加载程序进行解析。该表中将包含分区大小、闪存中的地址、RAM 中的加载地址、已加密/已签名等信息。针对每个分区（包含 FSBL）均存在一个此类结构。表中最后一个结构将全部标记 NULL 值（校验和除外）。下表显示了有关 AMD Zynq™ 7000 SoC 分区报头的偏移、名称和注释。

注释：含 3 个可加载节的 ELF 文件包含 1 个镜像报头表和 3 个分区报头表。

表 5：Zynq 7000 SoC 分区报头

偏移	名称	注释
0x00	Encrypted Partition length	已加密的分区数据长度。
0x04	Unencrypted Partition length	未加密的数据长度。
0x08	Total partition word length (Includes Authentication Certificate)。请参阅 Zynq 7000 SoC 身份验证证书 。	分区总字长由已加密的信息长度（含填充）、扩展长度和身份验证长度组成。
0x0C	Destination load address。	此分区要加载到的 RAM 地址。
0x10	Destination execution address。	此分区执行时的入口点。
0x14	Data word offset in Image	与启动镜像开始位置相关的分区数据的位置。
0x18	Attribute Bits	请参阅 Zynq 7000 SoC 分区属性位
0x1C	Section Count	单一分区内的节数。
0x20	Checksum Word Offset	启动镜像中对应校验和字的位置。
0x24	Image Header Word Offset	启动镜像中对应镜像报头的位置。
0x28	Authentication Certification Word Offset	启动镜像中对应身份验证证书的位置。
0x2C-0x38	Reserved	保留
0x3C	Header Checksum	按分区报头中的标准算法计算所得之前所有字的总和的反码。

Zynq 7000 SoC 分区属性位

下表描述了 AMD Zynq™ 7000 SoC 器件的分区报头表的分区属性位。

表 6：Zynq 7000 SoC 分区属性位

位字段	描述	注释
31:18	保留	不使用
17:16	分区所有者	0: FSBL 1: UBOOT 2 和 3: 保留
15	RSA 签名是否存在	0: 无 RSA 身份验证证书 1: RSA 身份验证证书

表 6: Zynq 7000 SoC 分区属性位 (续)

位字段	描述	注释
14:12	校验和类型	0: 无 1: MD5 2-7: 保留
11:8	保留	不使用
7:4	目标器件	0: 无 1: PS 2: PL 3: INT 4-15: 保留
3:2	保留	不使用
1:0	保留	不使用

Zynq 7000 SoC 身份验证证书

“Authentication Certificate”（身份验证证书）是包含分区身份验证相关所有信息的结构。此结构具有公钥和 BootROM/FSBL 需验证的所有签名。在每个“身份验证证书”内都包含一个“Authentication Header”（身份验证报头），可提供诸如密钥大小、用于签名的算法等信息。“身份验证证书”附加到启用身份验证的实际分区内。如果针对任一分区启用身份验证，那么报头表同样需进行身份验证。“报头表身份验证证书”将附加到报头表内容末尾。

AMD Zynq™ 7000 SoC 将 RSA-2048 身份验证与 SHA-256 散列算法结合使用，这表示主密钥和辅助密钥大小均为 2048 位。由于 SHA-256 用作为安全散列算法，因此 FSBL、分区和身份验证证书必须填充至 512 位边界。

AMD Zynq™ 7000 SoC 中的身份验证证书格式如下表所示。

表 7: Zynq 7000 SoC 身份验证证书

身份验证证书位	描述	
0x00	身份验证报头 = 0x0101000。请参阅 Zynq 7000 SoC 身份验证证书报头 。	
0x04	证书大小	
0x08	UDF (56 字节)	
0x40	PPK	模数 (256 字节)
0x140		模数扩展 (256 字节)
0x240		指数
0x244		填充 (60 字节)
0x280	SPK	模数 (256 字节)
0x380		模数扩展 (256 字节)
0x480		指数 (4 字节)
0x484		填充 (60 字节)
0x4C0	SPK 签名 = RSA-2048 (PSK, 填充 SHA-256 (SPK))	
0x5C0	FSBL 分区签名 = RSA-2048 (SSK, SHA256 (启动报头 FSBL 分区))	
0x5C0	其他分区签名 = RSA-2048 (SSK, SHA-256 (分区 填充 身份验证报头 PPK SPK SPK 签名))	

Zynq 7000 SoC 身份验证证书报头

下表描述了 AMD Zynq™ 7000 SoC 身份验证证书报头。

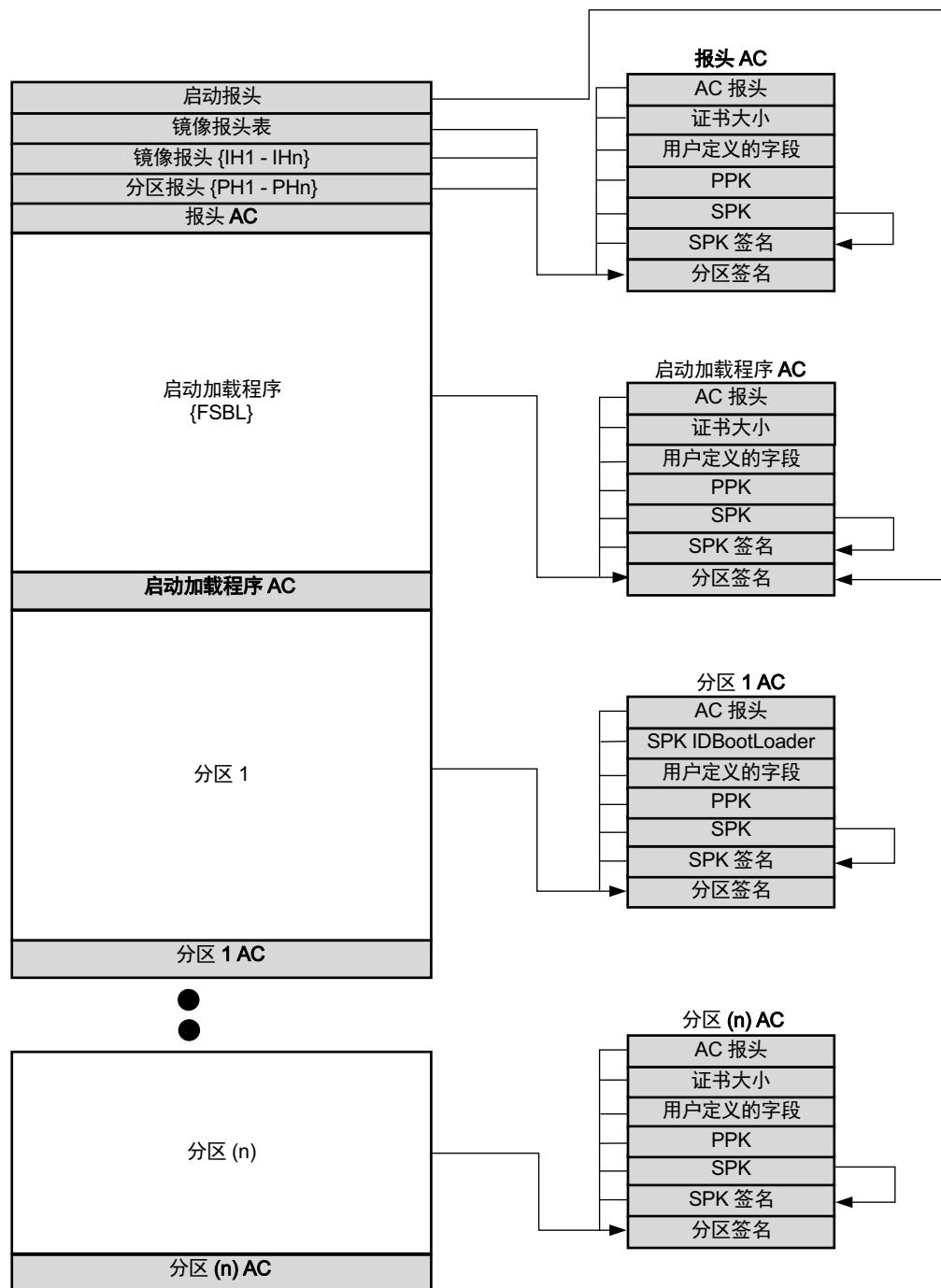
表 8：Zynq 7000 SoC 身份验证证书报头

位偏移	字段名称	描述
31:16	Reserved	0
15:14	Authentication Certificate Format	00: PKCS #1 v1.5
13:12	Authentication Certificate Version	00: 当前 AC
11	PPK Key Type	0: 散列密钥
10:9	PPK Key Source	0: eFUSE
8	SPK Enable	1: SPK 使能
7:4	Public Strength	0:2048
3:2	Hash Algorithm	0: SHA256
1:0	Public Algorithm	0: 保留 1: RSA 2: 保留 3: 保留

Zynq 7000 SoC 启动镜像模块框图

以下是可包含在 AMD Zynq™ 7000 SoC 启动镜像内的组件图示。

图 2：Zynq 7000 SoC 启动镜像模块框图



X30165-111824

Zynq UltraScale+ MPSoC 启动和配置

简介

AMD Zynq™ UltraScale+™ MPSoC 支持从不同器件启动，例如，QSPI 闪存、SD 卡、USB 设备固件升级 (DFU) 主机和 NAND 闪存设备等。本章详述了在安全模式和非安全模式下使用不同启动器件的启动流程。启动流程由平台管理单元 (PMU) 和配置安全单元 (CSU) 进行管理和执行。

初次启动期间将执行下列步骤：

- 通过上电复位 (POR) 使 PMU 解复位。
- PMU 执行来自 PMU ROM 的代码。
- PMU 初始化 SYSMON 和启动所需的 PLL、清空低功耗域和全功耗域，并释放 CSU 复位。

在 PMU 释放 CSU 后，CSU 将执行以下操作：

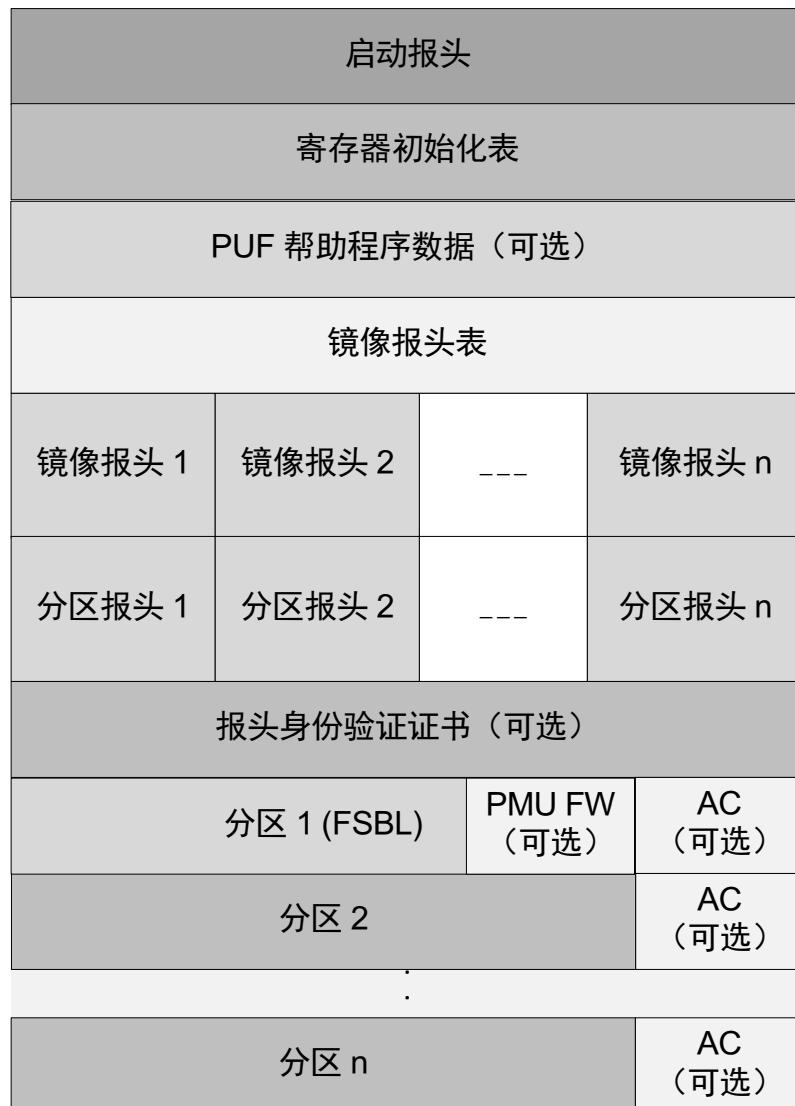
- 检查并判定 FSBL 或用户应用是否需要进行身份验证。
- 执行身份验证检查，并且只有在身份验证检查通过以后才继续执行。然后，检查镜像是否包含任何已加密的分区。
- 如果 CSU 检测到分区已加密，那么 CSU 会执行解密并初始化 OCM、判定启动模式设置、执行 FSBL 加载和可选 PMU 固件加载操作。
- 执行 CSU ROM 代码后，它会将控制权交给 FSBL。FSBL 使用 PCAP 接口来对含比特流的 PL 进行烧录。

随后，FSBL 会负责对系统进行操作。《Zynq UltraScale+ 器件技术参考手册》([UG1085](#)) 提供了有关 CSU 和 PMU 的详细信息。如需了解有关 CSU 的具体信息，请参阅《Zynq UltraScale+ MPSoC：软件开发指南》([UG1137](#)) 中的“配置安全性单元”。

Zynq UltraScale+ MPSoC 启动镜像

下图显示了 AMD Zynq™ UltraScale+™ MPSoC 启动镜像。

图 3: Zynq UltraScale+ MPSoC 启动镜像



X23449-102919

Zynq UltraScale+ MPSoC 启动报头

关于启动报头

Bootgen 将启动报头附加到任意启动镜像开头处。启动报头表为包含主启动加载程序（如 FSBL）相关启动信息的结构。此结构在整个启动镜像中唯一。该表由 BootROM 解析，以获取闪存中 FSBL 存储位置以及 OCM 中所需的 FSBL 加载位置的信息。其中还存储有部分加密和身份验证相关的参数。启动镜像组件包括：

- [Zynq UltraScale+ MPSoC 启动报头](#)，也包括 [Zynq UltraScale+ MPSoC 启动报头属性位](#)。
- [Zynq UltraScale+ MPSoC 寄存器初始化表](#)
- [Zynq UltraScale+ MPSoC PUF 帮助程序数据](#)
- [Zynq UltraScale+ MPSoC 镜像报头表](#)

- Zynq UltraScale+ MPSoC 镜像报头
- Zynq UltraScale+ MPSoC 身份验证证书
- Zynq UltraScale+ MPSoC 分区报头

在向 FSBL 交接控制权之前，BootROM 使用启动报头来查找 FSBL 的位置和长度信息以及有关对系统进行初始化的详细信息。下表提供了 AMD Zynq™ UltraScale+™ MPSoC 器件的地址偏移、参数和描述。

表 9：Zynq UltraScale+ MPSoC 器件启动报头

地址偏移	参数	描述
0x00-0x1F	Arm® vector table	XIP ELF 矢量表： 0xEFFFFFFE：对应于 Cortex®-R5F 和 Cortex A53 (32 位) 0x14000000：对应于 Cortex A53 (64 位)
0x20	Width Detection Word	该字段用于 QSPI 宽度检测。0xAA995566 (小字节序格式)。
0x24	Header Signature	包含 4 字节的 “X”、“N”、“L”、“X” (按字节顺序)，按小字节序格式为 0x584c4e58。
0x28	Key Source	0x00000000 (未加密) 0xA5C3C5A5 (eFUSE 中存储的黑密钥) 0xA5C3C5A7 (eFUSE 中存储的模糊密钥) 0x3A5C3C5A (BBRAM 中存储的红密钥) 0xA5C3C5A3 (eFUSE 中存储的 eFUSE 红密钥) 0xA35C7CA5 (启动报头中存储的模糊密钥) 0xA3A5C3C5 (启动报头中存储的用户密钥) 0xA35C7C53 (启动报头中存储的黑密钥)
0x2C	FSBL Execution address (RAM)	OCM 或 XIP 基址中的 FSBL 执行地址。
0x30	Source Offset	如无 PMUFW，则这是 FSBL 的起始偏移。如有 PMUFW，则为 PMUFW 的起始偏移。
0x34	PMU Image Length	PMU 固件原始镜像长度 (以字节为单位)。(0-128 KB)。 如果大小 > 0，PMUFW 将作为 FSBL 的前缀。 如果大小 = 0，则无 PMUFW 镜像。
0x38	Total PMU FW Length	PMUFW 镜像总长 (以字节为单位)。(PMUFW 长度 + 加密开销)
0x3C	FSBL Image Length	原始 FSBL 镜像长度 (以字节为单位)。(0-250 KB)。如为 0，则使用 XIP 启动镜像。
0x40	Total FSBL Length	FSBL 镜像长度 + FSBL 镜像的加密开销 + 身份验证。证书 + 64 字节对齐 + 散列大小 (完整性检查)。
0x44	FSBL Image Attributes	请参阅 位属性。
0x48	Boot Header Checksum	按标准算法，从偏移 0x20 到 0x44 (含) 计算所得的字数总和的反码。这些字假定按小字节序。
0x4C-0x68	Obfuscated/Black Key Storage	存储模糊密钥或黑密钥。
0x6C	Shutter Value	32 位 PUF_SHUT 寄存器值，用于配置 PUF 的快门偏移时间和快门打开时间。

表 9: Zynq UltraScale+ MPSoC 器件启动报头 (续)

地址偏移	参数	描述
0x70 - 0x94	User-Defined Fields (UDF)	40 字节。
0x98	Image Header Table Offset	指向镜像报头表的指针。
0x9C	Partition Header Table Offset	指向分区报头的指针。
0xA0-0xA8	Secure Header IV	用于启动加载程序分区的安全报头的 IV。
0xAC-0xB4	Obfuscated/Black Key IV	用于模糊密钥或黑密钥的 IV。

Zynq UltraScale+ MPSoC 启动报头属性位

表 10: Zynq UltraScale+ MPSoC 启动报头属性位

字段名称	位偏移	宽度	默认	描述
Reserved	31:16	16	0x0	保留。必须为 0。
BHDR RSA	15:14	2	0x0	0x3 - 对启动镜像执行 RSA 身份验证时，不执行 PPK 散列和 SPK ID 验证。 所有其他值：根据 eFUSE RSA 位来判定 RSA 身份验证。
Reserved	13:12	2	0x0	不适用
CPU Select	11:10	2	0x0	0x0: R5 单 CPU 0x1: A53 单 CPU 32 位 0x2: A53 单 CPU 64 位 0x3: R5 双 CPU
Hashing Select	9:8	2	0x0	0x0、0x1：无完整性检查 0x3：使用 SHA3 执行 BI 完整性检查
PUF-HD	7:6	2	0x0	0x3: PUF HD 包含在启动报头内。 所有其他值：PUF HD 包含在 eFUSE 中
Reserved	5:0	6	0x0	保留以供将来使用。必须为 0。

Zynq UltraScale+ MPSoC 寄存器初始化表

Bootgen 中的“寄存器初始化表”采用 256 个地址/值对格式，用于为 MIO 多路复用器和闪存时钟初始化 PS 寄存器。如需了解更多信息，请参阅 [初始化对和 INT 文件属性](#)。

表 11: Zynq UltraScale+ MPSoC 寄存器初始化表

地址偏移	参数	描述
0xB8 到 0x8B4	寄存器初始化对： <address>:<value>: (2048 字节)	如果“Address”设置为 0xFFFFFFFF, 则将跳过该寄存器并忽略该值。所有未使用的寄存器字段都必须设置为 Address=0xFFFFFFFF, value =0x0。

Zynq UltraScale+ MPSoC PUF 帮助程序数据

PUF 使用帮助程序数据在部件寿命有效期内，重新创建 KEK 原始值，该值可覆盖受保障工作温度和电压的完整范围。帮助程序数据由 <syndrome_value>、<aux_value> 和 <chash_value> 组成。帮助程序数据可存储在 eFUSE 或启动镜像中。如需了解更多信息，请参阅 [puf_file](#)。另请参阅《Zynq UltraScale+ 器件技术参考手册》(UG1085) 中的“PUF 帮助程序数据”部分。

表 12: Zynq UltraScale+ MPSoC PUF 帮助程序数据

地址偏移	参数	描述
0x8B8 到 0xEC0	PUF Helper Data (1544 bytes)	仅当启动报头偏移 0x44 (位 7:6) == 0x3 时才有效。如果未插入 PUF HD，则启动报头大小 = 2048 字节。如果已插入 PUF 报头数据，则启动报头大小 = 3584 字节。PUF HD 大小 = 总大小 = 1536 字节的 PUFHD + 4 字节的 CHASH + 2 字节的 AUX + 1 字节的对齐 = 1544 字节。

Zynq UltraScale+ MPSoC 镜像报头表

Bootgen 通过从 ELF 文件、比特流、数据文件等中提取数据来创建启动镜像。从中提取数据的这些文件被称为镜像。每个镜像都包含一个或多个分区。“Image Header”（镜像报头）表采用包含所有镜像公用的信息以及如下信息的结构：镜像数量、启动镜像中存在的分区数量以及指向其他报头表的指针。

表 13: Zynq UltraScale+ MPSoC 器件镜像报头表

地址偏移	参数	描述
0x00	Version	0x01010000 0x01020000 - 0x10 字段为添加的字段
0x04	Count of Image Header	表示镜像报头的数量。
0x08	First Partition Header Offset	指向首个分区报头（字偏移）的指针。
0x0C	First Image Offset Header	指向首个镜像报头（字偏移）的指针。
0x10	Header Authentication Certificate	指向报头身份验证证书（字偏移）的指针。

表 13: Zynq UltraScale+ MPSoC 器件镜像报头表 (续)

地址偏移	参数	描述
0x14	Secondary Boot Device	选项包括： 0 - 相同启动器件 1 - QSPI-32 2 - QSPI-24 3 - NAND 4 - SD0 5 - SD1 6 - SDLS 7 - EMMC/ MMC 8 - USB 9 - ETHERNET 10 - PCIE 11 - SATA
0x18 - 0x38	Padding	保留 (0x0)
0x3C	Checksum	按镜像报头中的标准算法计算所得之前所有字的总和的反码。

Zynq UltraScale+ MPSoC 镜像报头

关于镜像报头

“Image Header”（镜像报头）为阵列结构，其中包含每个镜像的相关信息，例如，ELF 文件、比特流、数据文件等。每个镜像都可具有多个分区，例如，每个 ELF 均可包含多个可加载节，每个节均构成启动镜像中的一个分区。该表还包含镜像相关分区数量的信息。下表提供了 AMD Zynq™ UltraScale+™ MPSoC 的地址偏移、参数和描述。

表 14: Zynq UltraScale+ MPSoC 器件镜像报头

地址偏移	参数	描述
0x00	Next image header offset	链接到下一个镜像报头。如果当前镜像报头为最后一个镜像报头，则为 0（字偏移）。
0x04	Corresponding partition header	链接到首个关联的分区报头（字偏移）。
0x08	保留	始终为 0。
0x0C	Partition Count	实际分区计数值。
0x10 - N	Image Name	以大字节序打包。为对该字符串进行重构，请每次解包 4 字节、反转顺序然后串联。例如，字符串“FSBL10.ELF”打包为 0x10: ‘L’ , ‘B’ , ‘S’ , ‘F’ , 0x14: ‘E’ , ‘.’ , ‘.’ , ‘0’ , ‘_’ , 0x18: ‘\0’ , ‘\0’ , ‘F’ , ‘.’ , ‘L’ 。打包后的镜像名称为 4 字节的倍数。
可变	String Terminator	0x000000
可变	Padding	默认为 0xFFFFFFFF，边界为 64 字节。

Zynq UltraScale+ MPSoC 分区报头

关于分区报头

“Partition Header”（分区报头）为阵列结构，其中包含每个分区的相关信息。每个分区报头表均由启动加载程序进行解析。该表中将包含分区大小、闪存中的地址、RAM 中的加载地址、已加密/已签名等信息。针对每个分区（包含 FSBL）均存在一个此类结构。表中最后一个结构将全部标记 NULL 值（校验和除外）。下表显示了有关 AMD Zynq™ UltraScale+™ MPSoC 的偏移、名称和注释。

表 15：Zynq UltraScale+ MPSoC 器件分区报头

偏移	名称	注释
0x0	Encrypted Partition Data Word Length	已加密的分区数据长度。
0x04	Un-encrypted Data Word Length	未加密的数据长度。
0x08	Total Partition Word Length (Includes Authentication Certificate)。请参阅 身份验证证书 。	加密 + 填充 + 扩展 + 身份验证的总长。
0x0C	Next Partition Header Offset	下一个分区报头的位置（字偏移）。
0x10	Destination Execution Address LO	加载后此分区的低位 32 位可执行地址。
0x14	Destination Execution Address HI	加载后此分区的高位 32 位可执行地址。
0x18	Destination Load Address LO	此分区要加载到的 RAM 地址的低位 32 位部分。
0x1C	Destination Load Address HI	此分区要加载到的 RAM 地址的高位 32 位部分。
0x20	Actual Partition Word Offset	与启动镜像开始位置相关的分区数据的位置。（字偏移）
0x24	Attributes	请参阅 Zynq UltraScale+ MPSoC 分区属性位
0x28	Section Count	与该属性关联的节数。
0x2C	Checksum Word Offset	启动镜像中校验和表的位置。（字偏移）
0x30	Image Header Word Offset	启动镜像中对应镜像报头的位置。（字偏移）
0x34	AC Offset	启动镜像中对应身份验证证书（如果存在）的位置（字偏移）
0x38	Partition Number/ID	分区 ID。
0x3C	Header Checksum	按分区报头中的标准算法计算所得之前所有字的总和的反码。

Zynq UltraScale+ MPSoC 分区属性位

下表描述了 AMD Zynq™ UltraScale+™ MPSoC 的分区报头表的分区属性位。

表 16：Zynq UltraScale+ MPSoC 器件分区属性位

位偏移	字段名称	描述
31:24	Reserved	
23	Vector Location	异常矢量的位置。 0: LOVEC（默认值） 1: HIVEC
22:20	Reserved	

表 16: Zynq UltraScale+ MPSoC 器件分区属性位 (续)

位偏移	字段名称	描述
19	Early Handoff	加载后立即交接： 0: 无提早交接 1: 启用提早交接
18	Endianness	0: 小字节序 1: 大字节序
17:16	Partition Owner	0: FSBL 1: U-Boot 2 和 3: 保留
15	RSA Authentication Certificate present	0: 无 RSA 身份验证证书 1: RSA 身份验证证书
14:12	Checksum Type	0: 无 1-2: 保留 3: SHA3 4-7: 保留
11:8	Destination CPU	0: 无 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: 保留
7	Encryption Present	0: 未加密 1: 已加密
6:4	Destination Device	0: 无 1: PS 2: PL 3-15: 保留
3	A5X Exec State	0: AARCH64 (默认值) 1: AARCH32

表 16: Zynq UltraScale+ MPSoC 器件分区属性位 (续)

位偏移	字段名称	描述
2:1	Exception Level	0: EL0 1: EL1 2: EL2 3: EL3
0	Trustzone	0: 无安全 1: 安全

Zynq UltraScale+ MPSoC 身份验证证书

“Authentication Certificate”（身份验证证书）是包含分区身份验证相关所有信息的结构。此结构具有公钥和 BootROM/FSBL 需验证的签名。在每个“身份验证证书”内都包含一个“Authentication Header”（身份验证报头），可提供诸如密钥大小、用于签名的算法等信息。“身份验证证书”附加到启用身份验证的实际分区内。如果针对任一分区启用身份验证，那么报头表同样需进行身份验证。“Header Table Authentication Certificate”（报头表身份验证证书）将附加到报头表内容末尾。

AMD Zynq™ UltraScale+™ MPSoC 使用 RSA-4096 身份验证，这表示主密钥和辅助密钥大小均为 4096 位。下表提供了 Zynq UltraScale+ MPSoC 器件的“身份验证”证书的格式。

表 17: Zynq UltraScale+ MPSoC 器件身份验证证书

身份验证证书		
0x00	身份验证报头 = 0x0101000。请参阅 Zynq UltraScale+ MPSoC 身份验证证书报头 。	
0x04	SPK ID	
0x08	UDF (56 字节)	
0x40	PPK	模数 (512)
0x240		模数扩展 (512)
0x440		指数 (4 字节)
0x444		填充 (60 字节)
0x480	SPK	模数 (512 字节)
0x680		模数扩展 (512 字节)
0x880		指数 (4 字节)
0x884		填充 (60 字节)
0x8C0	SPK 签名 = RSA-4096 (PSK, 填充 SHA-384 (SPK + 身份验证报头 + SPK-ID))	
0xAC0	启动报头签名 = RSA-4096 (SSK, 填充 SHA-384 (启动报头))	
0xCC0	分区签名 = RSA-4096 (SSK, 填充 SHA-384 (分区 填充 身份验证报头 SPK ID UDF PPK SPK SPK 签名 BH 签名))	

注释：FSBL 签名计算方式如下：

```
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 身份验证证书报头

下表描述了 AMD Zynq™ UltraScale+™ MPSoC 器件的身份验证报头位字段。

表 18：身份验证报头位字段

位字段	描述	注释
31:20	保留	0
19:18	SPK eFUSE/用户 eFUSE 选择	01: SPK eFUSE 10: 用户 eFUSE
17:16	PPK 密钥选择	0: PPK0 1: PPK1
15:14	身份验证证书格式	00: PKCS #1 v1.5
13:12	身份验证证书版本	00: 当前 AC
11	PPK 密钥类型	0: 散列密钥
10:9	PPK 密钥源	0: eFUSE
8	SPK 使能	1: SPK 使能
7:4	公钥强度	0: 保留 1: 4096 2:3: 保留
3:2	散列算法	1: SHA3/384 2:3: 保留
1:0	公用算法	0: 保留 1: RSA 2: 保留 3: 保留

Zynq UltraScale+ MPSoC 安全报头

选择对分区进行加密时，Bootgen 会向该分区追加安全报头。此安全报头包含用于对实际分区进行加密的 Key/IV。而此报头则使用器件密钥和 IV 来进行加密。下表中显示了 Zynq UltraScale+ MPSoC 安全报头。

图 4: Zynq UltraScale+ MPSoC 安全报头

AES

	分区 #0 (FSBL)				分区 #1				分区 #2			
	已加密 使用		内容		已加密 使用		内容		已加密 使用		内容	
安全报头	Key0	IV0	-	IV1_#0	Key0	IV0+0x01	Key1_#1	IV1_#1	Key0	IV0+0x02	Key1_#2	IV1_#2
块 #0	Key0	IV1_#0	-	-	Key1_#1	IV1_#1	-	-	Key1_#2	IV1_#2	-	-

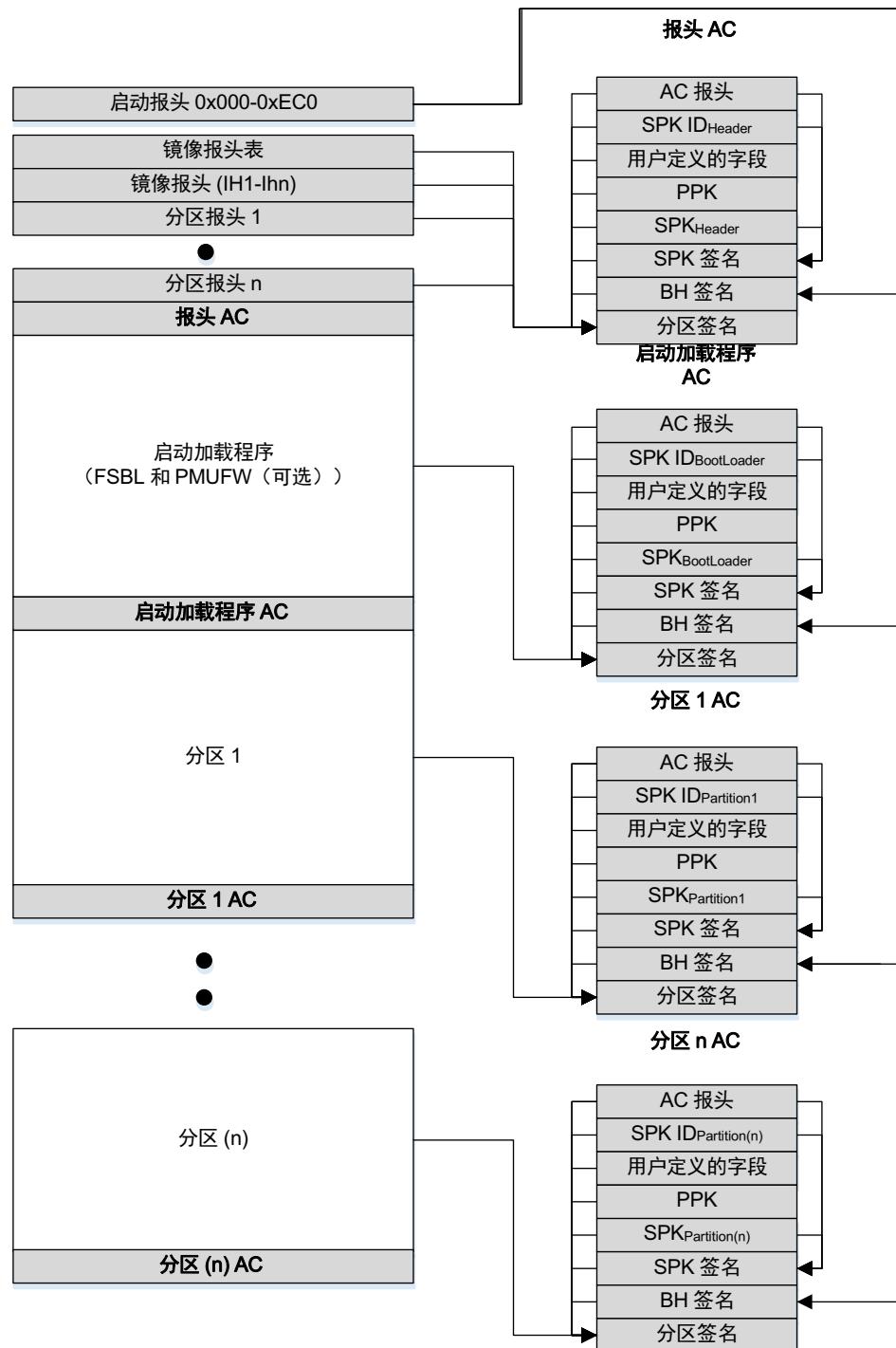
AES (含密钥滚动)

	分区 #0 (FSBL)				分区 #1				分区 #2			
	已加密 使用		内容		已加密 使用		内容		已加密 使用		内容	
安全报头	Key0	IV0	-	IV1_#0	Key0	IV0+0x01	Key1_#1	IV1_#1	Key0	IV0+0x02	Key1_#2	IV1_#2
块 #0	Key0	IV1_#0	Key2_#0	IV2_#0	Key1_#1	IV1_#1	Key2_#1	IV2_#1	Key1_#2	IV1_#2	Key2_#2	IV2_#2
块 #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
块 #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 启动镜像模块框图

以下是可包含在 AMD Zynq™ UltraScale+™ MPSoC 启动镜像内的组件图示。

图 5：Zynq UltraScale+ MPSoC 器件启动镜像模块框图



X18916-081518

Versal 自适应 SoC 启动镜像格式

以下是可包含在 AMD Versal™ 自适应 SoC 启动镜像（名为可编程器件镜像 (PDI)）中的组件图示。

平台管理控制器

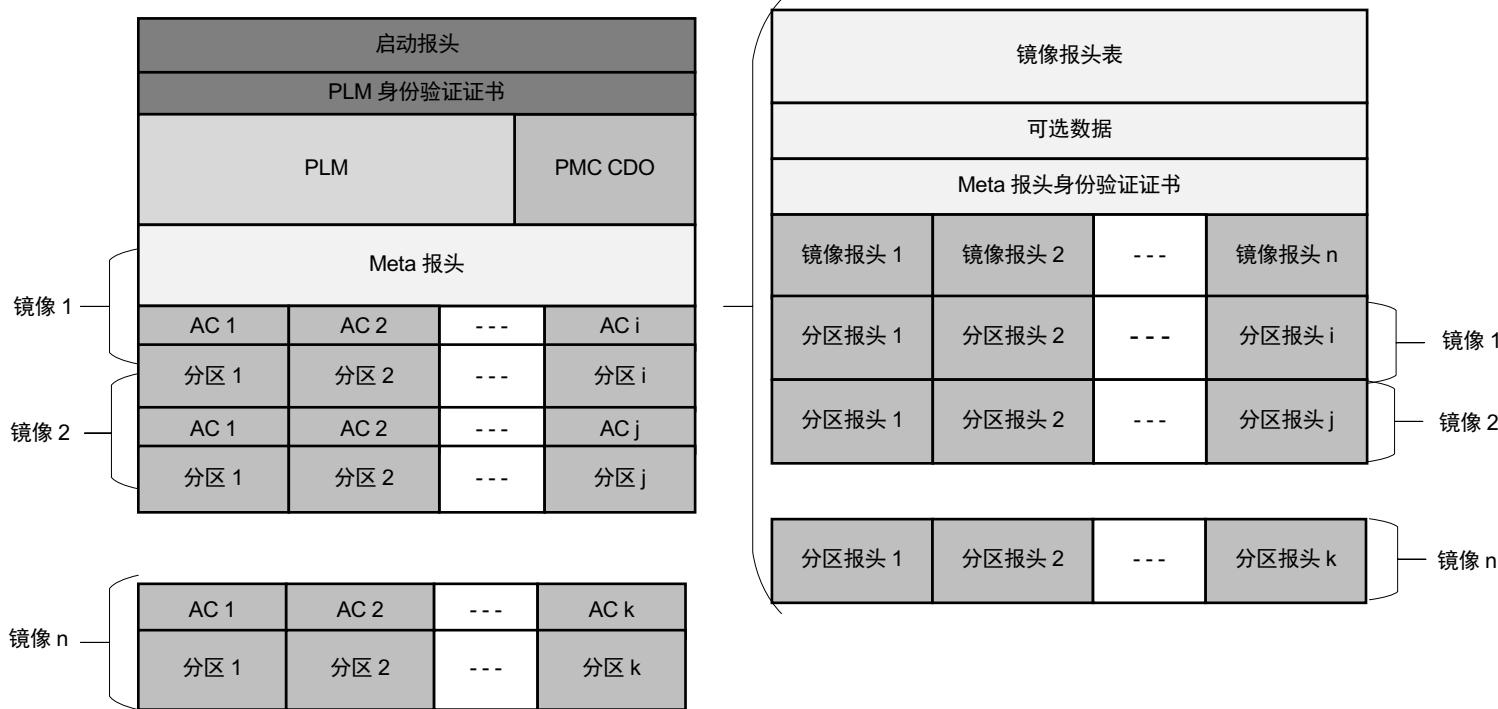
Versal 自适应 SoC 中的平台管理控制器 (PMC) 负责 Versal 自适应 SoC 的平台管理，包括启动和配置。本章主要讲解由 2 个 PMC MicroBlaze 处理器、ROM 代码单元 (RCU) 和平台处理单元 (PPU) 所处理的启动镜像格式：

- RCU：ROM 代码单元包含三重冗余 MicroBlaze 处理器和只读存储器 (ROM)，其中包含可执行 BootROM。BootROM 可执行文件采用金属屏蔽，不可更换。RCU 中的 MicroBlaze 处理器负责确认和运行 BootROM 可执行文件。RCU 还负责启动后的安全监控和物理不可克隆功能 (PUF) 的管理。
- PPU：平台处理单元包含三重冗余 MicroBlaze 处理器和 384 KB 的专用 PPU RAM。PPU 中的 MicroBlaze 负责运行 Platform Loader and Manager (PLM)。

在 Versal 自适应 SoC 中，自适应引擎 (PL) 由 rCDO 和 rNPI 文件组成。rCDO 文件主要包含 CFrame 数据以及 PL 和 NoC 功耗域初始化命令。rNPI 文件包含与 NPI 块相关的配置数据。NPI 块包含 NoC 元素：NMU、NSU、NPS、NCRB、DDR、XPHY、XPIO、GTY、MMCM 等。

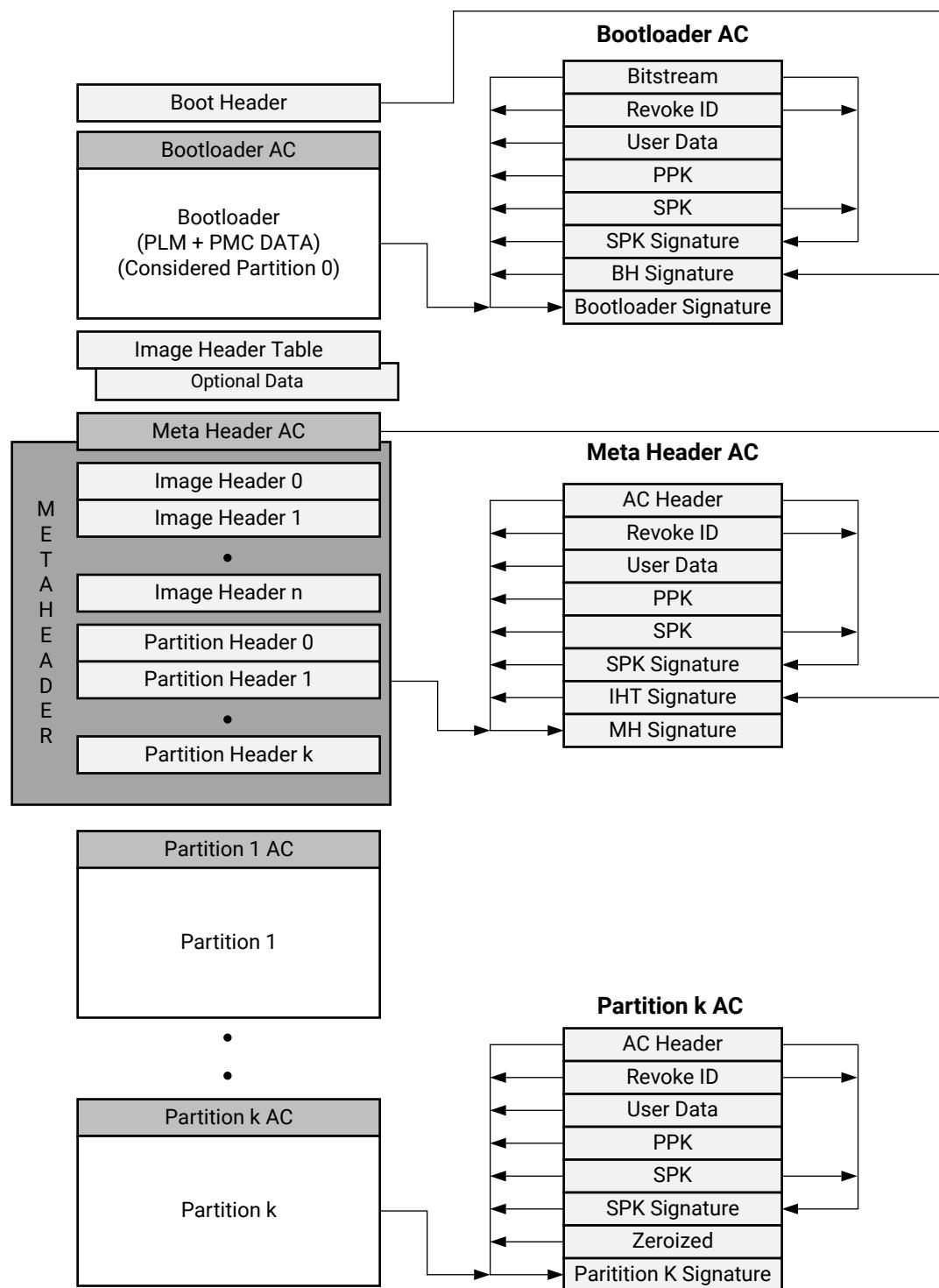
注释：AMD Versal™ 自适应 SoC 包含 SSI 技术器件。如需了解更多信息，请参阅 [第 6 章：SSIT 支持](#)。

图 6：Versal 自适应 SoC 启动镜像模块框图



X30208-120324

图 7：Versal 自适应 SoC 启动镜像模块框图第二部分



X28584-090823

Versal 自适应 SoC 启动报头

启动报头供 PMC BootROM 使用。根据启动报头中设置的属性，PMC BootROM 会确认 Platform Loader and Manager (PLM) 并将其加载到 PPU RAM 中。前 16 字节用于 SelectMAP 总线检测。PMC BootROM 和 PLM 会忽略此数据，因此 Bootgen 在其任意操作（如校验和、SHA、RSA、加密等）内都不包含此数据。以下代码片段是 SelectMAP 总线宽度探测码形位的示例。Bootgen 会根据所选宽度将以下数据置于前 16 字节内。

以下列表中显示了各镜像报头宽度和对应的位数：

- 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]

注释：默认 SelectMAP 宽度为 X32。

下表显示 AMD Versal™ 自适应 SoC 的启动报头格式。

表 19: Versal 自适应 SoC 启动报头格式

偏移 (十六进制)	大小 (字节)	描述	详细信息
0x00	16	SelectMAP 总线宽度	用于判定 SelectMAP 总线宽度为 x8、x16 还是 x32
0x10	4	QSPI 总线宽度	QSPI 总线宽度描述。此项是识别单堆模式、双堆模式或双并行模式下的 QSPI 闪存所必需的。0xAA995566 (小字节序格式)。
0x14	4	镜像标识	启动镜像标识字符串。包含 4 字节的 X、N、L、X (按字节顺序)，按小字节序格式为 0x584c4e58。
0x18	4	加密密钥源	该字段用于识别 AES 密钥源： 0x00000000 - 未加密 0xA5C3C5A3 - eFUSE 红密钥 0xA5C3C5A5 - eFUSE 黑密钥 0x3A5C3C5A - BBRAM 红密钥 0x3A5C3C59 - BBRAM 黑密钥 0xA35C7C53 - 启动报头黑密钥
0x1C	4	PLM 源偏移	PDI 中的 PLM 源偏移地址
0x20	4	PMC 数据加载地址	要加载的 PMC CDO 地址
0x24	4	PMC 数据长度	PMC CDO 长度
0x28	4	PMC 数据总长度	PMC CDO 长度 (包含身份验证和加密开销)
0x2C	4	PLM 长度	PLM 原始镜像大小
0x30	4	PLM 总长度	PLM 镜像大小 (包含身份验证和加密开销)
0x34	4	启动报头属性	Versal 自适应 SoC 启动报头属性
0x38	32	黑密钥	256 位密钥，仅当启动报头中的加密状态设置为黑密钥时才有效
0x58	12	黑 IV	解密黑密钥时使用的初始化矢量
0x64	12	安全报头 IV	安全报头初始化矢量

表 19: Versal 自适应 SoC 启动报头格式 (续)

偏移 (十六进制)	大小 (字节)	描述	详细信息
0x70	4	PUF 快门值	PUF 关闭快门前执行采样的时长 注释：此快门值必须与 PUF 寄存期间使用的快门值相匹配。
0x74	12	PMC 数据的安全报头 IV	用于对 PMC 数据的安全报头进行解密的 IV。
0x80	68	保留	以 0 填充。
0xC4	4	Meta 报头偏移	Meta 报头开始位置的偏移。
0xC8-0x124	96	保留	
0x128	2048	寄存器初始化	存储寄存器写入对，用于系统寄存器初始化
0x928	1544	PUF 帮助程序数据	PUF 帮助程序数据
0xF30	4	校验和	按标准算法计算所得报头校验和的反码。
0xF34	76	SHA3 填充	SHA3 标准填充

Versal 自适应 SoC 启动报头属性

下表中描述了镜像属性。

表 20: Versal 自适应 SoC 启动报头属性

字段名称	位偏移	宽度	默认值	描述
Reserved	[31:18]	14	0x0	保留以供将来使用。必须为 0
PUF Mode	[17:16]	2	0x0	0x3 - PUF 4K 模式。
Boot Header Authentication	[15:14]	2	0x0	0x3 - 对启动镜像执行身份验证时，不执行 PPK 散列和 SPK ID 验证。 所有其他值：根据 eFUSE RSA/ECDSA 位来判定身份验证。
Reserved	[13:12]	2	0x0	保留以供将来使用。必须为 0
DPA counter measure	[11:10]	2	0x0	0x3 - 启用 所有其他值均表示禁用 (eFUSE 会覆盖这些值)
Checksum selection	[9:8]	2	0x0	0x0、0x1 和 0x2 - 保留 0x3 - SHA3 用作为散列函数以执行校验和。
PUF HD	[7:6]	2	0x0	0x3 - PUF HD 包含在启动报头内 所有其他值 - PUF HD 包含在 eFUSE 内。
Reserved	[5:0]	6	0x0	保留

Versal 自适应 SoC 镜像报头表

下表包含 PDI 镜像相关的一般信息。

表 21：Versal 自适应 SoC 镜像报头表

偏移	名称	描述
0x0	Version	0x00040000(v4.0): 1. 添加针对 IHT 的 AAD 支持。 2. 在 32k 安全区块中已包含散列。 0x00030000(v3.0): 已将安全区块大小从 64 KB 更新为 32 KB 0x00020000(v2.00): IHT 和 PHT 大小翻倍
0x4	Total Number of Images	PDI 中的镜像总数
0x8	Image Header Offset	相对于首个镜像报头开始位置的字地址
0xC	Total Number of Partitions	PDI 中的分区总数
0x10	Partition Header Offset	相对于分区报头开始位置的字偏移
0x14	Secondary boot device address	表示辅助镜像存在的地址。 仅当属性中存在辅助启动器件时，此项才有效
0x1C	Image Header Table Attributes	请参阅 镜像报头表属性
0x20	PDI ID	用于识别 PDI
0x24	Parent ID	初始启动 PDI 的 ID。对于启动 PDI，它与 PDI ID 相同
0x28	Identification string	存在启动报头情况下的完整 PDI - FPDI 部分/子系统 PDI - PPDI
0x2C	Headers size	0-7: 镜像报头表大小（以字数为单位） 8-15: 镜像报头大小（以字数为单位） 16-23: 分区报头大小（以字数为单位） 24-31: 保留
0x30	Total meta header length (Word)	包含身份验证和加密开销（不包括 IHT，包括 AC）
0x34 - 0x3C	IV for encryption of meta header	用于解密报头表的 SH 的 IV
0x40	Encryption status	加密密钥源，对于 Meta 报头，仅限用于 PLM 的密钥源才有效。 0x00000000 - 未加密 0xA5C3C5A3 - eFUSE 红密钥 0xA5C3C5A5 - eFUSE 黑密钥 0x3A5C3C5A - BBRAM 红密钥 0x3A5C3C59 - BBRAM 黑密钥 0xA35C7C53 - 启动报头黑密钥
0x48	Meta Header AC Offset (Word)	Meta 报头身份验证证书的字偏移
0x4c	Meta Header Black/IV	用于对 Meta 报头加密所使用的黑密钥进行加密的 IV。
0x58	Optional Data Length (Word)	启动加载程序中可用的可选数据的大小
0x5C - 0x78	Reserved	0x0
0x7C	Checksum	按镜像报头表中的标准算法计算所得之前所有字的总和的反码

镜像报头表属性

下表中描述了镜像报头表。

表 22：Versal 自适应 SoC 镜像报头表属性

位字段	名称	描述
31:14	Reserved	0
14	PUF Helper Data Location	PUF 帮助程序数据 eFUSE/BH 的位置
12	dpacm enable	DPA 对策是否启用
11:6	Secondary boot device	<p>指示其他数据所在的器件。</p> <p>0 - 相同启动器件（默认）</p> <p>1 - QSPI32</p> <p>2 - QSPI24</p> <p>3 - 保留</p> <p>4 - SD0</p> <p>5 - SD1</p> <p>6 - SDLS</p> <p>7 - EMMC/ MMC</p> <p>8 - USB</p> <p>9 - 保留</p> <p>10 - PCIe</p> <p>11 - 保留</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>所有其他值均为保留值</p> <p>注释：针对 Bootgen 中的各种器件，支持使用这些选项。要获取任意器件支持的辅助启动器件的完整列表，请参阅其对应的系统软件开发者指南 (SSDG)。</p>
5:0		保留

可选数据

可选数据是置于 PDI 中的镜像报头表 (IHT) 之后的二进制数据。您可以加入自己的可选数据，如数据、版本等。如需了解更多信息，请参阅 [optionaldata](#)。在 IHT 签名过程中会对此数据进行身份验证，在加密期间，会将 IHT 作为镜像报头 (IH) 的首个安全报头 (SH) 的经过身份验证的额外数据 (AAD) 随 IHT 一并添加到其中。这意味着可选数据内容仍为纯文本。

作为身份验证优化的一部分，对于使用相同身份验证密钥的分区，分区散列将置于可选数据内。如需了解更多信息，请参阅 [身份验证优化](#)。

Bootgen 可接受用户数据和多种可选数据。如需了解有关添加可选数据的信息，请参阅 [optionaldata](#)。

表 23: Versal 自适应 SoC 可选数据

偏移	名称	描述
0x0	ID	来自可选数据 0x0 至 0x20 的数据 ID 均保留供内部使用。用户可选数据 ID 可采用 > 0x20 的任何 ID。 0 - 无, 可用于填充 1 - PLM 配置文档中定义的 PLM 构建时配置元数据 2 - 就地 PLM 更新兼容性检查中使用的数据结构版本信息 3 - 身份验证优化的散列表 4 - 表示存在 PSMFW
0x2	Size	可选数据总大小 (以码字为单位)
0x4	Data	实际数据 (包括填充)
末尾	Checksum	此数据结构中先前所有码字的总和

Versal 自适应 SoC 镜像报头

镜像报头为阵列结构，其中包含每个镜像的相关信息，例如，ELF 文件、CFrame、NPI、CDO、数据文件等。每个镜像都可具有多个分区，例如，每个 ELF 均可包含多个可加载节，每个节均构成启动镜像中的一个分区。镜像报头指向与该镜像关联的分区（分区报头）。在单一镜像内可使用 BIF 关键字“image”来组合多个分区文件；此方法适用于将单一公用子系统或功能的相关所有分区组合到一起。Bootgen 可为每个文件创建所需的分区，并为该镜像创建单一公用镜像报头。下表包含镜像相关分区数量的信息。

表 24: Versal 自适应 SoC 镜像报头

偏移	名称	描述
0x0	First Partition Header (Word)	首个分区报头的字偏移
0x4	Number of Partitions	此镜像中存在的分区数量
0x8	Revoke ID	Meta 报头的撤销 ID
0xC	Image Attributes	请参阅镜像属性表
0x10-0x1C	Image Name	镜像的 ASCII 名称。最多包含 16 个字符。需填充时，以 0 填充。
0x20	Image/Node ID	用于定义镜像正在初始化的资源节点
0x24	Unique ID	用于定义给定器件资源所需的亲和性/兼容性标识
0x28	Parent Unique ID	用于定义镜像的配置内容所需的父级资源 UID (如果需要)
0x2c	Function ID	此标识用于捕获镜像配置数据的唯一功能
0x30	DDR Low Address for Image Copy	下 32 位 DDR 地址，在 BIF 中启用 memcpy 时，必须将镜像复制到该地址
0.34	DDR High Address for Image Copy	上 32 位 DDR 地址，在 BIF 中启用 memcpy 时，必须将镜像复制到该地址
0x38	Reserved	
0x3C	Checksum	按标准算法计算所得之前所有字的总和的反码。

下表显示了镜像报头属性。

表 25：Versal 自适应 SoC 镜像报头属性

位字段	名称	描述
31:9	Reserved	0
8	Delay Hand off	0 - 立即交接镜像（默认） 1 - 稍后交接镜像
7	Delay load	0 - 立即加载镜像（默认） 1 - 稍后加载镜像
6	Copy to memory	0 - 不复制到存储器（默认） 1 - 镜像将复制到存储器
5:3	Image Owner	0 - PLM（默认值） 1 - 非 PLM 2-7 - 保留
2:0	Reserved	0

Versal 自适应 SoC 分区报头

分区报头包含有关分区的详细信息，如下表中所述。

表 26：Versal 自适应 SoC 分区报头表

偏移	名称	描述
0x0	Partition Data Word Length	加密分区数据长度
0x4	Extracted Data Word Length	未加密数据长度
0x8	Total Partition Word Length (Includes Authentication Certificate)	加密 + 填充 + 扩展 + 身份验证的总长
0xC	Next Partition header offset (Word)	下一个分区报头的偏移
0x10	Destination Execution Address (Lower Half)	加载后此分区的低位 32 位可执行地址。
0x14	Destination Execution Address (Higher Half)	加载后此分区的可执行地址的高位 32 位部分。
0x18	Destination Load Address (Lower Half)	此分区要加载到的 RAM 地址的低位 32 位部分。对于 ELF 文件，Bootgen 会从 ELF 格式自动读取。对于 RAW 数据，您必须指定其加载位置。对于 CFI 和配置数据，必须将其设为 0xFFFF_FFFF
0x1C	Destination Load Address (Higher Half)	此分区要加载到的 RAM 地址的高位 32 位部分。对于 ELF 文件，Bootgen 会从 ELF 格式自动读取。对于 RAW 数据，您必须指定其加载位置。对于 CFI 和配置数据，必须将其设为 0xFFFF_FFFF
0x20	Data Word Offset in Image	与启动镜像开始位置相关的分区数据的位置。
0x24	Attribute Bits	请参阅分区属性表
0x28	Section Count	如果镜像类型为 elf，则它会显示与此 elf 关联的其他分区的数量。
0x2C	Checksum Word Offset	启动镜像中校验和字的位置。
0x30	Partition ID	分区 ID

表 26: Versal 自适应 SoC 分区报头表 (续)

偏移	名称	描述
0x34	Authentication Certification Word Offset	启动镜像中身份验证证书的位置。
0x38 - 0x40	IV	IV 表示分区的安全报头。
0x44	Encryption Key select	加密状态： 0x00000000 - 未加密 0xA5C3C5A3 - eFUSE 红密钥 0xA5C3C5A5 - eFUSE 黑密钥 0x3A5C3C5A - BBRAM 红密钥 0x3A5C3C59 - BBRAM 黑密钥 0xA35C7C53 - 启动报头黑密钥 0xC5C3A5A3 - 用户密钥 0 0xC3A5C5B3 - 用户密钥 1 0xC5C3A5C3 - 用户密钥 2 0xC3A5C5D3 - 用户密钥 3 0xC5C3A5E3 - 用户密钥 4 0xC3A5C5F3 - 用户密钥 5 0xC5C3A563 - 用户密钥 6 0xC3A5C573 - 用户密钥 7 0x5C3CA5A3 - eFUSE 用户密钥 0 0x5C3CA5A5 - eFUSE 用户黑密钥 0 0xC3A5C5A3 - eFUSE 用户密钥 1 0xC3A5C5A5 - eFUSE 用户黑密钥 1
0x48	Black IV	IV 用于加密该分区的密钥源。
0x54	Revoke ID	分区撤销 ID
0x58-0x78	Reserved	0
0x7C	Header Checksum	按分区报头中的标准算法计算所得之前所有字的总和的反码。

下表列出了分区报头表属性。

表 27: Versal 自适应 SoC 分区报头表属性

位字段	名称	描述
31:29	Reserved	0x0
28:27	DPA CM Enable	0 - 禁用 1 - 启用
26:24	Partition Type	0 - 保留 1 - elf 2 - 配置数据对象 3 - Cframe 数据 (PL 数据) 4 - 原始数据 5 - 原始 elf 6 - CFI GSR CSC 反掩码帧 7 - CFI GSR CSC 掩码帧

表 27: Versal 自适应 SoC 分区报头表属性 (续)

位字段	名称	描述
23	HiVec	对应 RPU/APU (32 位) 处理器的 VInitHi 设置 0 - LoVec 1 - HiVec
22:19	Reserved	0
18	Endianness	0 - 小字节序 (默认值) 1 - 大字节序
17:16	Partition Owner	0 - PLM (默认值) 1 - 非 PLM 2.3 - 保留
15:14	PUF HD location	0 - eFUSE 1 - 启动报头
13:12	Checksum Type	000b - 无校验和 (默认值) 011b - SHA3
11:8	Destination CPU	0 - 无 (针对非 elf 文件, 这是默认值) 1 - A72-0 2 - A72-1 3 - 保留 4 - 保留 5 - R5-0 6 - R5-1 7 - R5-L 8 - PSM 9 - AIE 10-15 - 保留
4:7	Reserved	0x0
3	A72 CPU execution state	0 - Aarch64 (默认值) 1 - Aarch32
2:1	Exception level (EL) that the A72 core must be configured for	00b - EL0 01b - EL1 10b - EL2 11b - EL3 (默认值)
0	TZ secure partition	0 - 非安全 (默认值) 1 - 安全 此位表明 PLM 需配置的核 (需在其中执行此分区) 是否必须配置为 TrustZone 安全。默认情况下, 此项必须为 0。

Versal 自适应 SoC 身份验证证书

“Authentication Certificate” (身份验证证书) 是包含分区身份验证相关所有信息的结构。此结构具有公钥和 BootROM/PLM 需验证的签名。在每个 “身份验证证书” 内都包含一个 “Authentication Header” (身份验证报头)，可提供诸如密钥大小、用于签名的算法等信息。不同于其他器件，“身份验证证书” 将附加到启用身份验证的实际分区之前或之后。如果 Bootgen 要对 Meta 报头执行身份验证，请在 bif 属性 “metaheader” 下显式指定身份验证。请参阅 [附录 B: BIF 属性参考](#) 以获取用法信息。

Versal 自适应 SoC 使用 RSA-4096 身份验证和 ECDSA 算法来执行身份验证。下表提供了 Versal 自适应 SoC 的身份验证证书格式。

表 28：Versal 自适应 SoC 身份验证证书 - ECDSA p384

身份验证证书位		描述
0x00	身份验证报头。请参阅 Versal 自适应 SoC 身份验证证书报头	
0x04	撤销 ID	
0x08	UDF (56 字节)	
0x40	PPK	x (48 字节)
		y (48 字节)
		填充 0x00 (932 字节)
0x444	PPK SHA3 填充 (12 字节)	
0x450	SPK	x (48 字节)
		y (48 字节)
		填充 0x00 (932 字节)
0x854	SPK SHA3 填充 (4 字节)	
0x858	对齐 (8 字节)	
0x860	SPK 签名 (r+s+填充) (48+48+416)	
0xA60	BH/IHT 签名 (r+s+填充) (48+48+416)	
0xC60	分区签名 (r+s+填充) (48+48+416)	

表 29：Versal 自适应 SoC 身份验证证书 - ECDSA p521

身份验证证书位		描述
0x00	身份验证报头。请参阅 Versal 自适应 SoC 身份验证证书报头	
0x04	撤销 ID	
0x08	UDF (56 字节)	
0x40	PPK	PPK x (66 字节)
		y (66 字节)
		填充 0x00 (896 字节)
0x444	PPK SHA3 填充 (12 字节)	
0x450	SPK	SPK x (66 字节)
		y (66 字节)
		填充 0x00 (896 字节)
0x854	SPK SHA3 填充 (4 字节)	
0x858	对齐 (8 字节)	
0x860	SPK 签名 (r+s+填充) (66+66+380)	
0xA60	BH/IHT 签名 (r+s+填充) (66+66+380)	
0xC60	分区签名 (r+s+填充) (66+66+380)	

表 30: Versal 自适应 SoC 身份验证证书 - RSA

身份验证证书位		描述
0x00	身份验证报头。请参阅 Versal 自适应 SoC 身份验证证书报头	
0x04	撤销 ID	
0x08	UDF (56 字节)	
0x40	PPK	模数 (512 字节)
		模数扩展 (512 字节)
		指数 (4 字节)
0x444	PPK SHA3 填充 (12 字节)	
0x450	SPK	模数 (512 字节)
		模数扩展 (512 字节)
		指数 (4 字节)
0x854	SPK SHA3 填充 (4 字节)	
0x858	对齐 (8 字节)	
0x860	SPK 签名	
0xA60	BH/IHT 签名	
0xC60	分区签名	

Versal 自适应 SoC 身份验证证书报头

下表描述了 Versal 自适应 SoC 的身份验证报头位字段。

表 31：身份验证报头位字段

位字段	描述	注释
31:16	保留	0
15-14	身份验证证书格式	00 - RSAPSS
13-12	身份验证证书版本	00: 当前 AC
11	PPK 密钥类型	0: 散列密钥
10-9	PPK 密钥源	0: eFUSE
8	SPK 使能	1: SPK 使能
7-4	公钥强度	0 - ECDSA p384 1 - RSA 4096 2 - ECDSA p521
3-2	散列算法	1-SHA3

表 31：身份验证报头位字段 (续)

位字段	描述	注释
1-0	公用算法	1-RSA 2-ECDSA

注释：

1. 对于启动加载程序分区：
 - a. AC 的偏移 0xA60 包含启动报头签名。
 - b. AC 的偏移 0xC60 包含 PLM 和 PMCDATA 签名。
2. 对于报头表：
 - a. AC 的偏移 0xA60 包含 IHT 签名。
 - b. AC 的偏移 0xC60 包含除 IHT 外的所有报头的签名。
3. 对于任何其他分区：
 - a. AC 的偏移 0xA60 已补零。
 - b. AC 的偏移 0xC60 包含该分区的签名。

创建启动镜像

启动镜像格式 (BIF)

AMD 启动镜像布局具有多个文件、多种文件类型和多个支持报头用于供启动加载程序解析这些文件。Bootgen 定义了多个属性用于生成启动镜像，并根据文件中传递的内容来解释和生成启动镜像。由于有多条命令和多个属性可用，Bootgen 定义了启动镜像格式 (BIF) 来包含这些输入。BIF 包含：

- 配置属性，用于创建安全/非安全启动镜像
- 启动加载程序
 - 适用于 AMD Zynq™ 器件和 AMD Zynq™ UltraScale+™ MPSoC 的第一阶段启动加载程序 (FSBL)
 - 适用于 AMD Versal™ 自适应 SoC 的 Platform Loader and Manager (PLM)
 - **注释：**建议将相同发行版本的启动加载程序 (FSBL/PLM) 与 Bootgen 一起使用。
- 一个或多个分区镜像

Bootgen 利用多个属性和多条命令来定义创建启动镜像时的行为。例如，要为合格的 FPGA 器件、AMD Zynq™ 7000 SoC 器件、AMD Versal™ 自适应 SoC 系列、系列或 AMD Zynq™ UltraScale+™ MPSoC 器件创建启动镜像，必须向 Bootgen 提供相应的 [arch](#) 命令选项。以下附录列出并描述了用于指导 Bootgen 行为的可用选项。

- [附录 A: 用例与示例](#)
- [附录 B: BIF 属性参考](#)
- [附录 C: 命令参考](#)

启动镜像的格式遵循混用软硬件的格式要求。BootROM 加载程序需使用启动报头来加载单一分区（通常为启动加载程序）。启动镜像的其余部分由启动加载程序进行加载和处理。Bootgen 会通过组建分区列表来生成启动镜像。这些分区可包括：

- FSBL 或 PLM
- 第二阶段启动加载程序 (SSBL)，如 U-Boot
- 比特流 PL CFrame 数据、.rledo 和 .rnpi
- Linux
- 要在处理器上运行的软件应用
- 用户数据
- 由 Bootgen 生成的启动镜像。它可用于将新分区追加到先前生成的启动镜像中。

注释：请避免将工具版本与来自其他工具版本的初始 PDI 工件（如 PLM.elf、PSM.elf PMC/LPD/FPD.cdo）混用搭配。

注释：如需了解更多信息，请参阅《Vitis 统一软件平台文档：嵌入式软件开发》([UG1400](#))

BIF 语法和受支持的文件类型

BIF 文件用于指定启动镜像的每个组件（按启动顺序），并允许对每个镜像组件应用可选属性。在某些情况下，如果镜像组件在存储器中不连续，即可映射到多个分区。例如，如果某个 ELF 文件具有多个非连续的可加载节，那么每个节均可作为一个独立分区。BIF 文件语法规则如下：

```
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 = pmcdtata, load = 0xf2000000,  
            file = /path/to/pmc_cdo.bin  
        }  
    }  
}
```

注释：以上格式仅适用于 AMD Versal™，欲知详情，请参阅 [Versal 自适应 SoC 的 BIF 语法](#)。

```
<image_name>:  
{  
    // common attributes  
    [attribute1] <argument1>  
  
    // partition attributes  
    [attribute2, attribute3=<argument>] <elf>  
    [attribute2, attribute3=<argument>, attribute4=<argument>] <bit>  
    [attribute3] <elf>  
    <bin>  
}
```

- <image_name> 和 {...} 分组方式将要添加到 ROM 镜像的分区内的文件用括号括起。
- 在 {...} 括号内列出了一个或多个数据文件。
- 每个分区数据文件的数据文件名前均可包含一组可选属性，语法为 [attribute, attribute=<argument>]。
- 属性可用于对数据文件应用某些性质。
- 可使用 “,” 作为分隔符来分隔列示的多个属性。多个属性的顺序无关紧要。某些属性为单一关键字，某些为关键字等效字。
- 如果文件并非位于当前目录内，那么还可添加指向文件名的文件路径。文件以自由格式列示，全部位于一行上（以任意空格分隔，至少含一个空格），或者位于多行上。
- 空格将被忽略，可添加空格以保障可读性。
- 您可使用 C 语言样式的块注释：/*...*/，或者也可使用 C++ 行注释：//。

以下示例 BIF 文件含额外空格和换行，以提升可读性：

```
<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 支持的文件

下表列出了 Bootgen 支持的文件。

表 32：Bootgen 支持的文件

受支持的器件	扩展名	描述	注释
受所有器件支持	.bin	二进制	原始二进制文件。
	.dtb	二进制	原始二进制文件。
	image.gz	二进制	原始二进制文件。
	.elf	可执行链接文件 (ELF)	已移除符号和报头。
	.int	寄存器初始化文件	
	.nky	AES 秘钥	
	.pub/.pem	RSA 秘钥	
	.sig	签名文件	由 Bootgen 或 HSM 生成的签名文件。
Versal	.rledo	CFI 文件	仅适用于 Versal 器件。
	.cd0/.npi/.rnpi	CDO 文件	配置数据对象文件。仅适用于 Versal 器件。
	.bin/.pdi	启动镜像	使用 Bootgen 生成的启动镜像。
Zynq 7000/Zynq UltraScale+ MPSoC/FPGA	.bit/.rbt	比特流	剥离 BIT 文件头。

Versal 自适应 SoC 的 BIF 语法

以下示例演示了对分区进行分组时编写 BIF 的详细方式。为支持子系统概念，Versal 自适应 SoC 已更改 BIF 语法，在此类子系统中多个分区可组合在一起以形成一个镜像，这也称为含一个镜像报头的子系统。

注释：相同镜像 {} 块下的分区会合并以构成单个子系统。

```
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
        }
    }
}
```

以下示例简单演示了如何通过对分区进行分组来编写 BIF。

```
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.rpdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = gen_files/fpd_data.cdo }
    }
}
```

属性

下表列出了 Bootgen 属性。每个属性都包含 1 个链接，此链接指向左侧列中的详细描述以及右侧列中的简短描述。架构名称用于指示使用该属性的 AMD 器件：

- zynq: Zynq 7000 SoC 器件
- zynqmp: AMD Zynq™ UltraScale+™ MPSoC
- fpga: 任意 7 系列和更高版本的器件

- `versal`: AMD Versal™ 自适应 SoC

如需了解更多信息，请参阅 [附录 B: BIF 属性参考](#)。

表 33: Bootgen 属性和描述

选项/属性	描述	适用于
<code>aarch32_mode</code>	指定将以 32 位模式执行的二进制文件。	· <code>zynqmp</code> · <code>versal</code>
<code>aeskeyfile <aes_key_filepath></code>	指向 AES 密钥文件的路径。密钥文件包含用于加密分区的 AES 密钥。此密钥文件的内容需写入 eFUSE 或 BBRAM。如果指定路径内不存在密钥文件，则将由 Bootgen 生成新密钥用于加密。例如：如果针对 BIF 文件内的比特流选择了加密，那么输出即为已加密的比特流。	· 全部
<code>alignment <byte></code>	设置字节对齐。对此分区进行填充以对齐到该值的倍数。该属性不能配合偏移使用。	· <code>zynq</code> · <code>zynqmp</code>
<code>auth_params <options></code>	其他身份验证选项： <ul style="list-style-type: none"> · <code>ppk_select</code>: 针对受支持的 2 个 PPK, 0=1, 1=2。 · <code>spk_id</code>: 32 位 ID, 用于区分 SPK。 · <code>spk_select</code>: 用于区分 spk 和用户 eFUSE。默认值为 <code>spk-efuse</code>。 · <code>header_auth</code>: 在不对分区进行身份验证的情况下用于对报头进行身份验证。 	· <code>zynqmp</code>
<code>authentication <option></code>	指定要进行身份验证的分区。 <ul style="list-style-type: none"> · Zynq 的身份验证是使用 RSA-2048 完成的。 · Zynq UltraScale+ MPSoC 的身份验证是使用 RSA-4096 完成的。 · Versal 自适应 SoC 是使用 RSA-4096、ECDSA-p384 和 ECDSA-p521 进行身份验证的。 实参包括： <ul style="list-style-type: none"> · <code>none</code>: 分区未签名。 · <code>ecdsa-p384</code>: 分区已使用 ecdsa-p384 曲线签名 · <code>ecdsa-p521</code>: 分区已使用 ecdsa-p521 曲线签名 · <code>rsa</code>: 分区已使用 RSA 算法签名。 	· 全部
<code>bbram_kek_iv <filename></code>	指定用于加密对应密钥的 IV。 <code>bbram_kek_iv</code> 搭配 <code>keysr=bbram_blk_key</code> 使用时即为有效。	· <code>versal</code>
<code>bh_kek_iv <filename></code>	指定用于加密对应密钥的 IV。 <code>bh_kek_iv</code> 搭配 <code>keysr=bh_blk_key</code> 使用时即为有效。	· <code>versal</code>
<code>bh_key_iv <filename></code>	解密模糊密钥或黑密钥时使用的初始化矢量。	· <code>zynqmp</code>
<code>bh_keyfile <filename></code>	将存储在启动报头中的 256 位模糊密钥或黑密钥。仅当用于加密的 <code>keysr</code> 为 <code>bh_gry_key</code> 或 <code>bh_blk_key</code> 时，此项才有效。 注释：Versal 器件不支持模糊密钥。	· <code>zynqmp</code> · <code>versal</code>
<code>bhsignature <filename></code>	将启动报头签名导入身份验证证书。如果您不愿意共享秘密密钥 PSK，则可使用此项。您可创建签名并将其提供给 Bootgen。文件格式为 <code>botheader.sha384.sig</code> 。	· <code>zynqmp</code> · <code>versal</code>
<code>big_endian</code>	指定二进制文件为大字节序格式。	· <code>zynqmp</code> · <code>versal</code>
<code>blocks <block sizes></code>	指定加密中的密钥滚动功能的块大小。每个模块均使用其专用密钥进行加密。初始密钥存储在器件上的密钥源中，而每个后续块的密钥则在前一个模块中进行加密（封装）。	· <code>versal</code>

表 33：Bootgen 属性和描述 (续)

选项/属性	描述	适用于
<code>boot_config <options></code>	该属性可指定用于配置启动镜像的参数。	<ul style="list-style-type: none">· <code>versal</code>
<code>boot_device <options></code>	<p>指定辅助启动器件。指示分区所在的器件。选项包括：</p> <ul style="list-style-type: none">· <code>qspi32</code>· <code>qspi24</code>· <code>nand</code>· <code>sd0</code>· <code>sd1</code>· <code>sd-1s</code>· <code>mmc</code>· <code>usb</code>· <code>ethernet</code>· <code>pcie</code>· <code>sata</code>· <code>ospi</code>· <code>smap</code>· <code>sbi</code>· <code>sd0-raw</code>· <code>sd1-raw</code>· <code>sd-1s-raw</code>· <code>mmc-raw</code>· <code>mmc0</code>· <code>mmc0-raw</code> <p>注释：针对 Bootgen 中的各种器件，支持使用这些选项。要获取辅助启动选项的列表，请参阅《Versal 自适应 SoC 系统软件开发者指南》(UG1304) 或《Zynq UltraScale+ MPSoC：软件开发指南》(UG1137)。如需了解硬件/寄存器/接口信息和主启动模式，请参阅对应的 TRM，例如，《Zynq UltraScale+ 器件技术参考手册》(UG1085)、《Versal 自适应 SoC 技术参考手册》(AM011) 或《Versal 自适应 SoC 寄存器参考资料》(AM012)。</p>	<ul style="list-style-type: none">· <code>zynqmp</code>· <code>versal</code>
<code>bootimage <filename.bin></code>	指定所列举的输入文件是由 Bootgen 创建的启动镜像。	<ul style="list-style-type: none">· <code>zynq</code>· <code>zynqmp</code>· <code>versal</code>
<code>bootloader <partition></code>	指定分区为启动加载程序 (FSBL/PLM)。该属性随其他分区 BIF 属性一起指定。	<ul style="list-style-type: none">· <code>zynq</code>· <code>zynqmp</code>· <code>versal</code>
<code>bootvectors <vector_values></code>	指定就地执行 (XIP) 的矢量表。	<ul style="list-style-type: none">· <code>zynqmp</code>

表 33：Bootgen 属性和描述 (续)

选项/属性	描述	适用于
<code>checksum <options></code>	<p>指定需进行校验和的分区。不支持将该选项与更安全的功能（如身份验证和加密）一起使用。校验和算法为：</p> <ul style="list-style-type: none"> none：不执行校验和操作。 md5：仅限 AMD Zynq™ 7000 SoC 器件。 sha3：适用于 AMD Zynq™ UltraScale+™ MPSoC 器件和 Versal 器件。 <p>注释：Zynq 器件不支持对启动加载程序执行校验和。以下器件可支持对启动加载程序执行校验和操作：</p> <ul style="list-style-type: none"> Zynq UltraScale+ MPSoC Versal 自适应 SoC 	<ul style="list-style-type: none"> zynq zynqmp versal
<code>copy <address></code>	该属性可指定将镜像复制到位于指定地址的存储器。	<ul style="list-style-type: none"> versal
<code>core <options></code>	该属性用于指定执行分区的核。AMD Versal™ 自适应 SoC 的选项包括： <ul style="list-style-type: none"> a72-0 a72-1 r5-0 r5-1 psm aie r5-lockstep 	<ul style="list-style-type: none"> versal
<code>delay_handoff</code>	该属性可指定延迟交接至子系统/镜像。	<ul style="list-style-type: none"> versal
<code>delay_load</code>	该属性可指定延迟加载子系统/镜像。	<ul style="list-style-type: none"> versal
<code>delay_auth</code>	表示在后续阶段执行身份验证。这有助于 Bootgen 在分区加密期间保留空间以供散列使用。	<ul style="list-style-type: none"> versal
<code>destination_device <device_type></code>	指定分区目标为 PS 还是 PL。选项为： <ul style="list-style-type: none"> ps：分区目标为 PS（默认值）。 pl：分区目标为 PL（适用于比特流）。 	<ul style="list-style-type: none"> zynqmp
<code>destination_cpu <device_core></code>	指定应执行分区的核。 <ul style="list-style-type: none"> a53-0 a53-1 a53-2 a53-3 r5-0 (default) r5-1 pmu r5-lockstep 	<ul style="list-style-type: none"> zynqmp
<code>early_handoff</code>	此标志可确保加载分区后立即交接至关键应用；否则，将首先按顺序先加载所有分区，然后同样按顺序执行交接。	<ul style="list-style-type: none"> zynqmp
<code>efuse_kek_iv <filename></code>	指定用于加密对应密钥的 IV。efuse_kek_iv 搭配 keysrc=efuse_blk_key 使用时即为有效。	<ul style="list-style-type: none"> versal
<code>efuse_user_kek0_iv <filename></code>	指定用于加密对应密钥的 IV。efuse_user_kek0_iv 搭配 keysrc=efuse_user_blk_key0 使用时即为有效。	<ul style="list-style-type: none"> versal
<code>efuse_user_kek1_iv <filename></code>	指定用于加密对应密钥的 IV。efuse_user_kek1_iv 搭配 keysrc=efuse_user_blk_key1 使用时即为有效。	<ul style="list-style-type: none"> versal

表 33：Bootgen 属性和描述 (续)

选项/属性	描述	适用于
<code>encryption <option></code>	指定要加密的分区。加密算法为：zynq 使用 AES-CBC，而 zynqmp 和 Versal 则使用 AES-GCM。 分区选项为： <ul style="list-style-type: none">· none：分区不执行加密。· aes：分区使用 AES 算法执行加密。	· 全部
<code>exception_level <options></code>	核应配置为所示异常级别。 选项包括： <ul style="list-style-type: none">· el-0· el-1· el-2· el-3	· zynqmp · versal
<code>familykey <key file></code>	指定族密钥。	· zynqmp · fpga
<code>file <path/to/file></code>	该属性可指定用于创建分区的文件。	· versal
<code>fsbl_config <options></code>	指定用于配置启动镜像的子属性。这些子属性包括： <ul style="list-style-type: none">· <code>bh_auth_enable</code>：对启动镜像执行 RSA 身份验证时，不执行 PPK 散列和 SPK ID 验证。· <code>auth_only</code>：启动镜像仅使用 RSA 签名。FSBL 不应进行解密。· <code>opt_key</code>：使用运行密钥进行 block-0 解密。安全报头具有运行密钥。· <code>pufhd_bh</code>：PUF 帮助程序数据存储在启动报头中（默认值为 <code>efuse</code>）。· 使用 <code>[puf_file]</code> 选项将 PUF 帮助程序数据文件传递到 Bootgen。· <code>puf4kmode</code>：PUF 调整为在 4k 位配置内使用。· <code>shutter = <value></code>：32 位 <code>PUF_SHUT</code> 寄存器值，用于配置 PUF 的快门偏移时间和快门打开时间。 <p>注释：此快门值必须与 PUF 寄存期间使用的快门值相匹配。</p>	· zynqmp
<code>headersignature <signature_file></code>	将报头签名导入身份验证证书。如果您不想共享秘密密钥，则可使用此项。您可创建签名并将其提供给 Bootgen。	· zynq · zynqmp · versal
<code>hivec</code>	指定异常矢量表的位置为 hivec（高矢量）。默认值为 lovec（低矢量）。这仅适用于 A53 (32 位) 和 R5 核。 <ul style="list-style-type: none">· <code>hivec</code>：异常矢量表位于 <code>0xFFFF0000</code>。· <code>lovec</code>：异常矢量表位于 <code>0x00000000</code>。	· zynqmp
<code>id <id></code>	该属性可基于其定义位置来指定以下 ID： <ul style="list-style-type: none">· <code>pdi id</code> - 在最外层/PDI 括号内· <code>image id</code> - 在镜像括号内	· versal
<code>image</code>	定义子系统/镜像。	· versal
<code>init <filename></code>	位于启动加载程序末尾的寄存器初始化块，通过解析启动 (.int) 文件规格来构建。允许最多 256 个地址/值对。启动文件具有特定格式。	· zynq · zynqmp · versal

表33：Bootgen 属性和描述(续)

选项/属性	描述	适用于
<code>keysrc</code>	指定 Versal 自适应 SoC 的加密的密钥源。可针对各分区单独指定 keysrc。 <ul style="list-style-type: none"> · <code>efuse_red_key</code> · <code>efuse_blk_key</code> · <code>bbram_red_key</code> · <code>bbram_blk_key</code> · <code>bh_blk_key</code> · <code>user_key0</code> · <code>user_key1</code> · <code>user_key2</code> · <code>user_key3</code> · <code>user_key4</code> · <code>user_key5</code> · <code>user_key6</code> · <code>user_key7</code> · <code>efuse_user_key0</code> · <code>efuse_user_blk_key0</code> · <code>efuse_user_key1</code> · <code>efuse_user_blk_key1</code> 	· <code>versal</code>
<code>keysrc_encryption</code>	指定加密的密钥源。密钥包括： <ul style="list-style-type: none"> · <code>efuse_gry_key</code>: eFUSE 中存储的灰（模糊）密钥。请参阅 灰密钥/模糊密钥 · <code>bh_gry_key</code>: 启动报头中存储的灰（模糊）密钥。 · <code>bh_blk_key</code>: 启动报头中存储的黑密钥。请参阅 黑密钥/PUF 密钥 · <code>efuse_blk_key</code>: eFUSE 中存储的黑密钥。 · <code>kup_key</code>: 用户密钥。 · <code>efuse_red_key</code>: eFUSE 中存储的红密钥。请参阅 密钥滚动。 · <code>bbram_red_key</code>: BBRAM 中存储的红密钥。 	· <code>zynq</code> · <code>zynqmp</code>
<code>load <address></code>	为存储器内的分区设置期望的加载地址。	· <code>zynq</code> · <code>zynqmp</code> · <code>versal</code>
<code>metaheader</code>	该属性用于定义 Meta 报头的加密和身份验证属性，例如，密钥、密钥源等。	· <code>versal</code>
<code>name <name></code>	该属性可指定镜像/子系统的名称。	· <code>versal</code>
<code>offset <offset></code>	用于设置启动镜像中的分区的绝对偏移。	· <code>zynq</code> · <code>zynqmp</code> · <code>versal</code>
<code>optionaldata {<filename>, id=<id>}</code>	允许您指定数据 ID 和数据文件。	· <code>versal</code>
<code>parent_id</code>	该属性可指定父 PDI 的 ID。用于识别部分 PDI 与其对应启动 PDI 之间的关系。	· <code>versal</code>
<code>partition</code>	该属性用于定义分区。这是可选属性，用于简化 BIF 并使其可读。	· <code>versal</code>

表 33：Bootgen 属性和描述 (续)

选项/属性	描述	适用于
partition_owner 和 owner <option>	负责加载分区的分区的所有者。选项包括： 对于 Zynq/Zynq UltraScale+ MPSoC： <ul style="list-style-type: none">· fsbl：分区由 FSBL 加载。· uboot：分区由 U-Boot 加载。 对于 Versal： <ul style="list-style-type: none">· plm：分区由 PLM 加载。· non-plm：分区并非由 PLM 加载，而是由其他实体（如 U-Boot）加载。	<ul style="list-style-type: none">· zynq· zynqmp· versal
pid <ID>	指定分区 ID。PID 可为 32 位值（0 到 0xFFFFFFFF）。	<ul style="list-style-type: none">· zynqmp
pmufw_image <image_name>	PMU 固件镜像将由 BootROM 加载（在加载 FSBL 前）。	<ul style="list-style-type: none">· zynqmp
ppkfile <key filename>	主公钥 (PPK)。用于对启动镜像中的分区进行身份验证。 如需了解更多信息，请参阅 使用身份验证 。	<ul style="list-style-type: none">· zynq· zynqmp· versal
presign <sig_filename>	分区签名 (.sig) 文件。	<ul style="list-style-type: none">· zynq· zynqmp· fpga
pskfile <key filename>	主秘密密钥 (PSK)。用于对启动镜像中的分区进行身份验证。 如需了解更多信息，请参阅 使用身份验证 。	<ul style="list-style-type: none">· zynq· zynqmp· versal
puf_file <filename>	PUF 帮助程序数据文件。PUF 配合黑密钥用作为加密密钥源。 PUF 帮助程序数据为 1544 字节。其中 1536 个字节用于 PUF HD + 4 字节用于 HASH + 3 字节用于 AUX + 1 字节用于对齐。	<ul style="list-style-type: none">· zynqmp· versal
reserve <size in bytes>	保留存储器，分区后填充。	<ul style="list-style-type: none">· zynq· zynqmp· versal
spk_select <SPK_ID>	指定用户 eFUSE 中的 SPK ID。	<ul style="list-style-type: none">· zynqmp
spkfile <filename>	用于对启动镜像中的分区进行身份验证的密钥。如需了解更多信息，请参阅 使用身份验证 。	<ul style="list-style-type: none">· 全部
spksignature <signature_file>	将 SPK 签名导入身份验证证书。请参阅 使用身份验证 。您不愿意共享秘密密钥 PSK 时可使用此项。您可以创建签名并将其提供给 Bootgen。	<ul style="list-style-type: none">· zynq· zynqmp· versal

表 33：Bootgen 属性和描述 (续)

选项/属性	描述	适用于
<code>split <options></code>	<p>根据模式将镜像拆分为多个部分。拆分选项包括：</p> <ul style="list-style-type: none"> · Slaveboot：仅受 Zynq UltraScale+ MPSoC 支持。按如下方式拆分： · 启动报头 + 启动加载程序 · 镜像和分区报头 · 其他分区 · normal：受 zynq、zynqmp 和 versal 支持。按如下方式拆分： · 启动报头 + 镜像报头 + 分区报头 + 启动加载程序 · Partition1 · Partition2 以此类推 <p>除拆分模式外，输出格式还可指定为 <code>bin</code> 或 <code>mcs</code>。</p> <p>注释：选项拆分模式 <code>normal</code> 与命令行选项 <code>split</code> 相同。此命令行选项已被弃用。仅限 Zynq UltraScale+ MPSoC 才支持 <code>Split ulaveboot</code>。</p>	<ul style="list-style-type: none"> · zynq · zynqmp · versal
<code>sskfile <key filename></code>	辅助秘密密钥 (SSK) 用于对启动镜像中的分区进行身份验证。主密钥用于对辅助密钥进行身份验证；辅助密钥用于对分区进行身份验证。	· 全部
<code>startup <address></code>	加载分区后，设置其输入地址。针对不执行的分区忽略此项。	<ul style="list-style-type: none"> · zynq · zynqmp · versal
<code>trustzone <option></code>	TrustZone 选项为：	<ul style="list-style-type: none"> · secure · nonsecure
<code>type <options></code>	该属性用于指定分区类型。选项为：	· versal
	<ul style="list-style-type: none"> · bootloader · pmcdata · cdo · cfi · cfi-gsc · bootimage · slr-boot · slr-config 	
<code>udf_bh <data_file></code>	将要复制的数据文件导入启动报头的用户定义字段 (UDF)。UDF 通过十六进制字符串格式的文本文件来提供。UDF 中的字节总数为：zynq = 76 字节；zynqmp = 40 字节。	<ul style="list-style-type: none"> · zynq · zynqmp
<code>udf_data <data_file></code>	将包含最多 56 字节数据的文件导入身份验证证书的用户定义字段 (UDF)。	<ul style="list-style-type: none"> · zynq · zynqmp
<code>userkeys <filename></code>	指向用户密钥文件的路径。	· versal
<code>xip_mode</code>	指示直接从 QSPI 闪存执行 FSBL 的 eXecute In Place (XIP)。	<ul style="list-style-type: none"> · zynq · zynqmp

使用 Bootgen GUI

Bootgen 具有 GUI 和命令行选项。GUI 选项在 AMD Vitis™ IDE 中以向导形式提供。Vitis IDE 中的功能仅限于创建启动镜像时的大部分标准功能。但 Bootgen 命令行则可提供一整套全功能命令，以支持您为自己的系统创建复杂的启动镜像。

启动 Bootgen GUI

2024.1 中引入的 Vitis Unified IDE 使用 [Theia 环境](#)，但 Vitis 传统 IDE 则使用 Eclipse。在 Vitis 传统 IDE 与 Vitis Unified IDE 中，Bootgen 功能完全相同。但 GUI 布局有所不同。在本节中探讨了 Vitis Unified IDE 和 Vitis 传统 IDE 的 GUI 流程。

要使用 AMD Vitis™ 传统 IDE 和 Unified IDE 创建启动镜像，请执行以下任一操作：

- 对于 Vitis Unified IDE，在“Vitis Components”（Vitis 组件）视图中高亮应用组件，然后在“Flow”（流程）视图中，选中“Create Boot Image”（创建启动镜像）。
- 对于 Vitis 传统 IDE，在“Project Navigator”（工程导航器）或“C/C++ Projects”（C/C++ 工程）视图中选中应用工程，然后右键单击“Create Boot Image”。
- 或者，针对 Vitis 传统 IDE 和 Unified IDE，单击“Vitis” → “Create Boot Image” → “Target Device Family”（Vitis > 创建启动镜像 > 目标器件家族）。

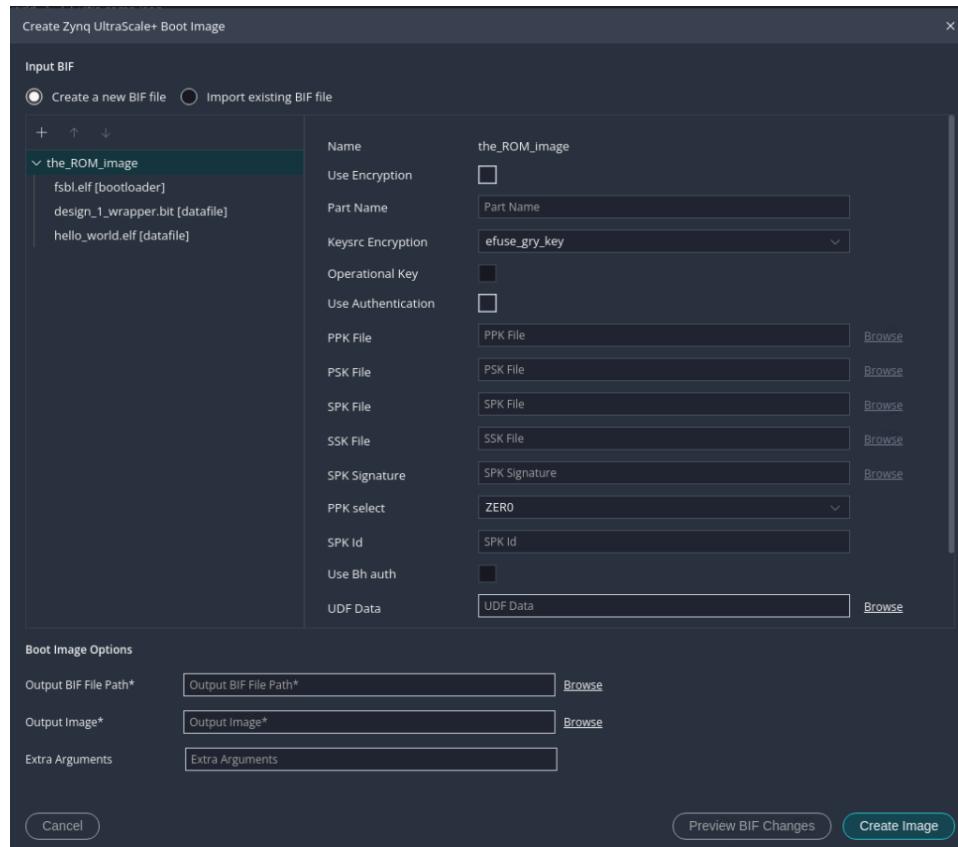
受支持的器件家族包括：

- AMD Zynq™ 和 Zynq AMD UltraScale+™
- AMD Versal™

适用于 Zynq 7000 器件和 Zynq UltraScale+ 器件的 Bootgen GUI

为 AMD Zynq™ 和 Zynq AMD UltraScale+™ 启动 Bootgen GUI 后，就会打开“Create Boot Image”（创建启动镜像）对话框，并且其中包含从所选工程上下文中预选的默认值。

图 8：在 Vitis Unified IDE 中为 Zynq 器件和 Zynq UltraScale+ 器件创建启动镜像



- 针对应用首次运行“Create Boot Image”时，在此对话框中将预填充 FSBL ELF 文件路径、所选硬件的比特流（如果在硬件工程中存于此比特流）以及所选应用 ELF 文件。
 - 如果先前为此应用运行了启动镜像，并且存在 BIF 文件，那么在此对话框中将预填充来自 /bif 文件夹的值。
1. 在“Create boot image”对话框中填充以下信息：
 - 从“Architecture”（架构）下拉菜单中选择所需的架构。
 - 选择“Create a BIF file”（创建 BIF 文件）或“Import an existing BIF file”（导入现有 BIF 文件）。
 - 在“Basic”（基础）选项卡中，指定“Output BIF file path”（输出 BIF 文件路径）。
 - 如果适用，请指定“UDF data”（UDF 数据）：请参阅 [udf_data](#) 以获取有关该选项的更多信息。
 - 指定“Output path”（输出路径）。
 2. 在“Boot image partitions”（启动镜像分区）中，单击“Add”（添加）按钮以添加其他分区镜像。
 3. 为启动镜像中的分区创建偏移值、对齐值和分配值（如适用）。
默认情况下，输出文件路径设置为所选应用工程下的 /bif 文件夹。
 4. 在“Security”（安全）选项卡中，可指定用于创建安全镜像的属性。此安全选项可根据需要应用于各分区。
 - 要为分区启用身份验证，请选中“Use Authentication”（使用身份验证）选项，然后指定 PPK、SPK、PSK 和 SSK 值。如需了解更多信息，请参阅 [使用身份验证](#) 主题。
 - 要为分区启用加密，请选中“Encryption”（加密）视图，然后选中“Use Encryption”（使用加密）选项。如需了解更多信息，请参阅 [使用加密](#)。

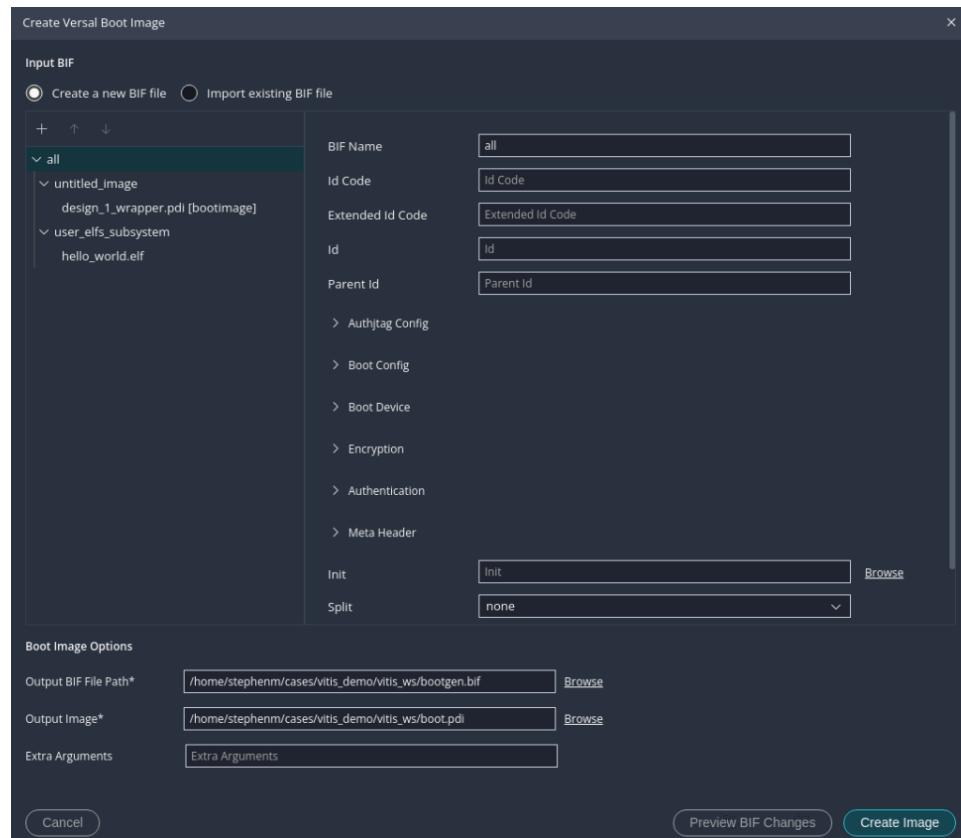
5. 逐一为每个分区创建或导入 BIF 文件启动镜像，从启动加载程序开始。分区列表可显示 BIF 文件中的分区摘要信息。其中可显示文件路径、加密设置和身份验证设置。此区域可用于对分区进行添加、删除、修改和重新排序。您还可设置启用加密、身份验证及校验和的值，并指定其他分区相关的值，如“Load”（加载）、“Alignment”（对齐）和“Offset”（偏移）

为 Versal 自适应 SoC 使用 Bootgen GUI 选项

从 Flow Navigator 创建 Versal 启动镜像

从 Flow Navigator 为 AMD Versal™ 启动 Bootgen GUI 后，就会打开“Create Boot Image”（创建启动镜像）对话框，并且其中包含从所选工程上下文中预选的默认值。

图 9：Vitis Unified IDE 中的 Versal 自适应 SoC 的 Bootgen GUI



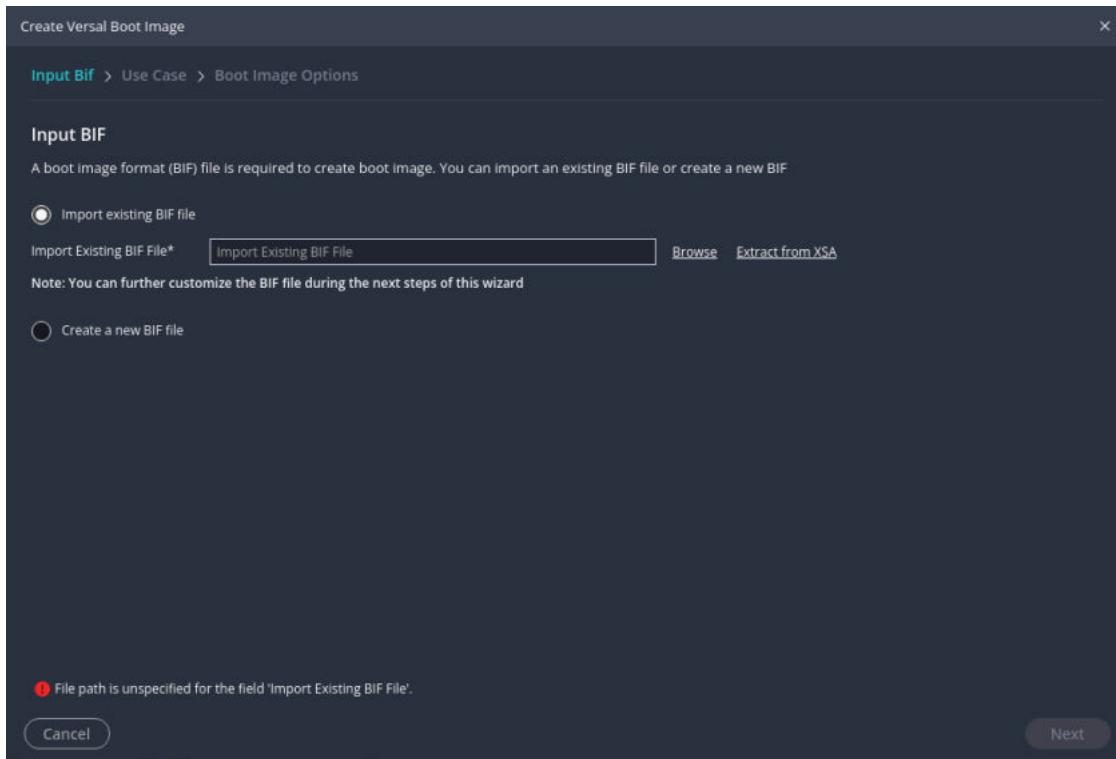
1. 在“Create boot image”对话框中填充以下信息：
 - a. 选择“Create a BIF file”（创建 BIF 文件）或“Import an existing BIF file”（导入现有 BIF 文件）。
注释：AMD Vivado™ 生成的 BIF 可在 <design>.runs/impl_1/ 目录中找到。
 - b. 在“Basic”（基础）选项卡中，指定“Output BIF file path”（输出 BIF 文件路径）。
 - c. 指定“Output Image”（输出镜像）。
2. 在“Boot image partitions”（启动镜像分区）中，单击“Add”（添加）按钮以添加其他分区镜像。

3. 为启动镜像中的分区创建偏移值、对齐值和分配值（如适用）。
默认情况下，输出文件路径设置为所选应用工程下的 `/bif` 文件夹。
4. 在“Security”（安全）选项卡中，可指定用于创建安全镜像的总体属性。选中“partition”（分区）并单击“Edit”（编辑）按钮，然后转至“Security”（安全）选项卡后，即可更新“Partition security”（分区安全）属性。
如需了解更多信息，请参阅[使用身份验证](#) 和 [使用加密](#)。
5. 逐一为每个分区创建或导入 BIF 文件启动镜像，从启动加载程序开始。分区列表可显示 BIF 文件中的分区摘要信息。其中可显示文件路径、加密设置和身份验证设置。此区域可用于对分区进行添加、删除、修改和重新排序。您还可设置启用加密、身份验证及校验和的值，并指定其他分区相关的值，如“Load”（加载）、“Alignment”（对齐）和“Offset”（偏移）
6. “Extra Bif attributes”（附加 Bif 属性）对话框中的内容以及“Edit partition” → “Extra Partition attributes”（编辑分区 > 附加分区属性）中的内容都会追加到总体 BIF 文件或分区。您可以使用这些文件来添加 Bootgen GUI 不支持的定制属性。

创建 Versal 启动镜像的常见用例

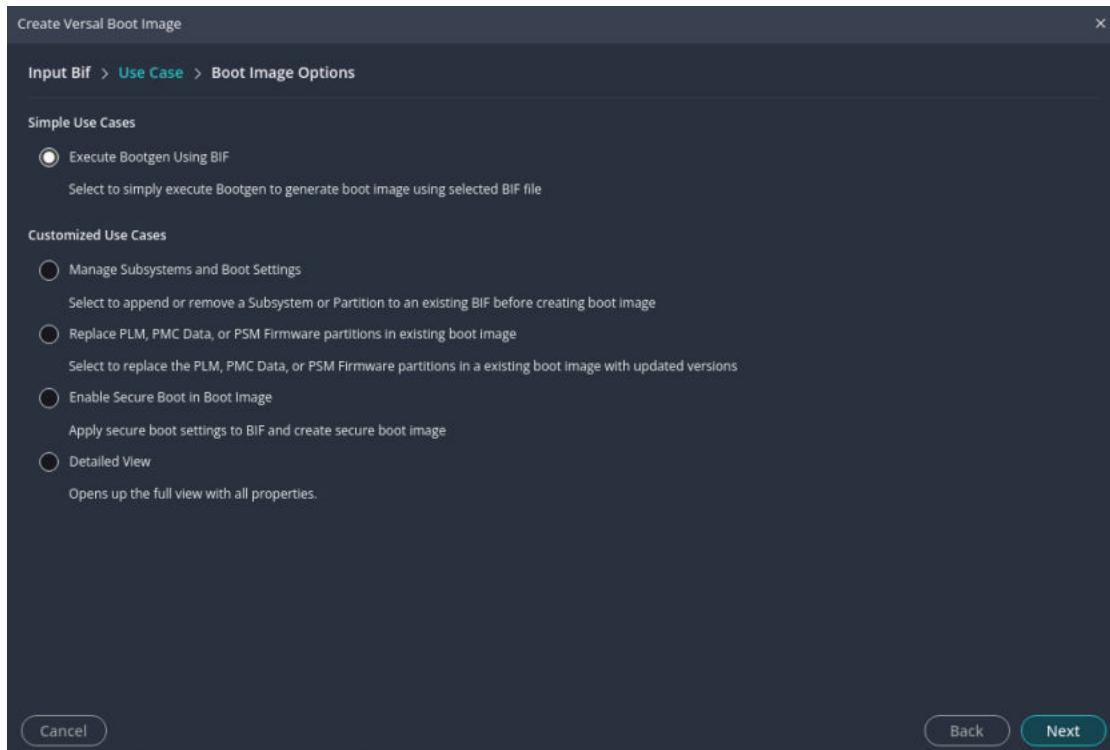
从 Vitis 菜单为 AMD Versal™ 启动 Bootgen GUI，这样即可打开“Create Boot Image”。

图 10：创建启动镜像



选择导入现有 BIF 文件，既可以使用现有的 BIF 文件，也可以从 XSA 文件中提取以获得该文件。然后就会出现快速创建启动镜像对话框的常见用例。

图 11：创建 Versal 启动镜像



客户可转至对应自定义用例，并快速创建启动镜像。

在命令行上使用 Bootgen

在命令行上指定 Bootgen 选项时，可供使用的选项远多于 Vitis IDE 中提供的选项。在 Vitis 软件平台的标准安装中，XSDB 可用作为交互式命令行环境或者用于创建脚本。在 XSDB 中，可运行 Bootgen 命令。XSDB 可访问 Bootgen 可执行文件，后者是一个独立工具。此 Bootgen 可执行文件可单独安装，如 [安装 Bootgen](#) 中所述。XSDB 中同样包含此工具，因此，该工具中开发的所有脚本均可在 XSDB 中运行，反之亦然。

命令和描述

下表中列出了 Bootgen 命令选项。每个选项都链接至左侧列中的详细描述以及右侧列中的简短描述。架构名称用于指示使用该命令的 AMD 器件：

- `zynq`: AMD Zynq™ 7000 SoC 器件
- `zynqmp`: AMD Zynq™ UltraScale+™ MPSoC
- `fpga`: 任意 7 系列和更高版本的器件
- `versal`: Versal 自适应 SoC

如需了解更多信息，请参阅 [附录 C: 命令参考](#)。

表 34：Bootgen 命令和描述

命令	描述和选项	适用于
<code>arch <type></code>	AMD 器件架构。选项： · zynq (默认) · zynqmp · fpga · versal	· 全部
<code>authenticatedjtag <options></code>	用于在安全启动期间启用 JTAG。实参包括： · rsa · ecdsa	· versal
<code>bif_help</code>	打印出 BIF 帮助信息摘要。	· 全部
<code>dual_qspi_mode <configuration></code>	生成 2 个输出文件用于双 QSPI 配置： · parallel · stacked <size>	· zynq · zynqmp · versal
<code>dual_ospf_mode stacked <size></code>	生成 2 个输出文件用于堆叠配置。	· versal
<code>dump <options></code>	根据指定的选项转储分区或启动报头。 · empty：将分区转储为二进制文件。 · bh：将启动报头转储为二进制文件。 · plm：将 PLM 转储为二进制文件。 · pmc_cdo：将 PMC CDO 转储为二进制文件。 · boot_files：将启动报头、PLM 和 PMC CDO 转储为三个独立二进制文件。 · slave_pdbs：为 SSI 技术用例转储从 PDI。 默认不对从 PDI 执行转储。如果您要单独对从 PDI 进行调试或分析，则应使用该选项。	· versal
<code>dump_dir</code>	将组件转储到指定目录中。	· versal
<code>efuseppkbits <PPK_filename></code>	生成用于 eFUSE 的 PPK 散列。	· zynq · zynqmp · versal
<code>enable_auth_opt</code>	用于启用身份验证优化	· versal
<code>encrypt <options></code>	器件中存储的 AES 密钥。选项包括： · bbram (默认) · efuse	· zynq · fpga
<code>encryption_dump</code>	生成加密 log 日志文件 <code>aes_log.txt</code> 。	· zynqmp · versal
<code>fill <hex_byte></code>	指定用于填充的填充字节。	· zynq · zynqmp · versal
<code>generate_hashes</code>	生成包含填充的散列的文件： · Zynq 器件：SHA-2 (含 PKCS#1v1.5 填充方案) · Zynq UltraScale+ MPSoC：SHA-3 (含 PKCS#1v1.5 填充方案) · Versal 自适应 SoC：SHA-3 (含 PSS 填充方案)	· zynq · zynqmp · versal

表 34：Bootgen 命令和描述 (续)

命令	描述和选项	适用于
<code>generate_keys <key_type></code>	生成认证密钥。选项包括： · pem · rsa · obfuscatedkey	· zynq · zynqmp · versal
<code>h 和 help</code>	打印出帮助信息摘要。	· 全部
<code>image <filename(.bif)></code>	提供启动镜像格式 (.bif) 文件名。	· 全部
<code>log<level_type></code>	生成位于当前工作目录的 log 日志文件，此文件含下列消息类型： · error · warning (默认) · info · debug · trace	· 全部
<code>nonbooting</code>	创建中间启动镜像。	· zynq · zynqmp · versal
<code>o <filename></code>	指定输出文件。文件格式取决于文件名扩展名。有效扩展名包括： · .bin (默认) · .mcs · .pdi	· 全部
<code>overlay_cdo <filename></code>	CDO 覆盖选项提供了生成 CDO 文件后对其进行修改的方法。	versal
<code>p <partname></code>	指定生成加密密钥时使用的部件名称。	· 全部
<code>padimageheader <option></code>	填充镜像报头，以强制对齐下列分区。选项包括： · 0 · 1 (默认值)	· zynq · zynqmp
<code>process_bitstream <option></code>	指定比特流作为 .bin 或 .mcs 来进行处理并输出。 · 例如：如果针对 BIF 文件中的比特流选中加密，那么输出即为已加密的比特流。	· zynq · zynqmp
<code>read <options></code>	用于根据选项读取启动报头、镜像报头和分区报头。 · bh：从启动镜像读取人工可读格式的启动报头 · iht：从启动镜像读取镜像报头表。 · ih：从启动镜像读取镜像报头。 · pht：从启动镜像读取分区报头。 · ac：从启动镜像读取身份验证证书	· zynq · zynqmp · versal
<code>split <options></code>	将启动镜像拆分为多个分区，并将文件作为 .bin 或 .mcs 来输出。 · 启动报头 + 镜像报头 + 分区报头 + Fsbl.elf · Partition1.bit · Partition2.elf	· zynq · zynqmp · versal

表 34：Bootgen 命令和描述 (续)

命令	描述和选项	适用于
<code>spksignature <filename></code>	生成 SPK 签名文件。	<ul style="list-style-type: none">· zynq· zynqmp· versal
<code>verify <filename></code>	该选项用于对启动镜像的身份验证执行验证。根据可用分区对启动镜像中的所有身份验证证书进行验证。	<ul style="list-style-type: none">· zynq· zynqmp· versal
<code>verify_kdf</code>	该选项用于确认 Bootgen 中用于生成 AES 密钥的“计数器模式 KDF”。	<ul style="list-style-type: none">· zynqmp· versal
<code>w <option></code>	指定是否覆盖输出文件： <ul style="list-style-type: none">· on (默认)· off 注释： 无选项的 -w 解释为 -w on。	<ul style="list-style-type: none">· 全部
<code>zynqmpes1</code>	为 ES1 (1.0) 生成启动镜像。默认填充方案为 ES2 (2.0)。	<ul style="list-style-type: none">· zynqmp

启动时间安全

AMD 支持在所有器件上使用最新身份验证方法来保障安全启动，防止在 AMD 器件上运行未经授权的代码或经修改的代码。AMD 支持各种加密技巧，以确保只有经授权的程序才能访问镜像。对于器件的硬件安全性功能，请参阅如下各章节。

Zynq 7000 SoC 器件中的安全模式和非安全模式

出于安全考量，在 PS 中的所有主模块中，CPU 0 始终为成功解复位的首个器件。CPU 1 置于 WFE 状态。运行 BootROM 时，JTAG 始终处于禁用状态以确保安全，与复位类型无关。BootROM 运行完成后，如果启动模式为非安全模式，则会启用 JTAG。

BootROM 代码还负责加载 FSBL/用户代码。当 BootROM 释放控制权并转至阶段 1 时，您的软件即可获得对整个系统的完整控制权。重新执行 BootROM 的唯一途径是生成任一系统复位。FSBL/用户代码大小限制为 192 KB，无论是否加密都是如此。此限制不适用于非安全就地执行选项。

PS 启动源是使用 BOOT_MODE 捆绑管脚（以弱上拉或下拉电阻来表示）选择的，这些管脚在上电复位 (POR) 期间进行采样。采样的值存储在 slcr.BOOT_MODE 寄存器中。

BootROM 支持将已加密/已验证镜像和未加密镜像分别称为安全启动镜像和非安全启动镜像。BootROM 支持使用就地执行 ([xip_mode](#)) 选项时直接从 NOR 或 QSPI 执行阶段 1 镜像，但这仅限于非安全启动镜像。就地执行仅适用于 NOR 和 QSPI 启动模式。

- 在安全启动模式下，运行 BootROM 代码的 CPU 会对启动器件上的用户 PS 镜像进行解密和身份验证、将其存储在 OCM 中，然后建立指向该镜像的分支。
- 在非安全模式下，运行 BootROM 代码的 CPU 会禁用包括 PL 内的 AES 单元在内的所有安全启动功能，然后再建立指向 OCM 内存或闪存器件（前提是使用就地执行 (XIP)）中的用户镜像的分支。

PS 或 PL 的任何后续启动阶段均由您负责，控制权归您所有。您无权访问 BootROM 代码。完成阶段 1 安全启动后，后续可执行安全或非安全启动阶段。以非安全模式下完成第一阶段启动后，后续只能执行非安全启动阶段。

Zynq UltraScale+ MPSoC 器件安全性

在 AMD Zynq™ UltraScale+™ MPSoC 器件中，安全启动是使用可信启动机制的硬件根完成的，这种方法也可用于加密所有启动文件或配置文件。此架构可提供用于托管最安全的应用所需的保密性、完整性和身份验证机制。

如需了解更多信息，请参阅《Zynq UltraScale+ 器件技术参考手册》([UG1085](#)) 的“安全性”部分。

Versal 自适应 SoC 安全

在 AMD Versal™ 自适应 SoC 上，安全启动可确保加载到器件上的固件和软件的保密性、完整性和身份验证机制。信任根随 PMC ROM 启动，后者可用于对 PLM 软件进行身份验证和/或解密。鉴于 PLM 软件可信，PLM 能以安全的方式来处理其余固件和软件的加载操作。此外，如果不需要安全启动，那么至少可采用简单的校验和来对软件加以确认。

如需了解更多信息，请参阅《Versal 自适应 SoC 技术参考手册》([AM011](#))。另请参阅《Versal 自适应 SoC 安全手册》(UG1508)。本手册需要从“设计安全性专区”下载有效的 NDA。

使用加密

安全启动已成为现场部署的大部分电子器件的必备功能，它会先确认器件上的镜像，然后再允许其执行。对于加密，AMD 支持高级加密标准 (AES) 算法 AES 加密。

AES 可提供对称密钥加密技术（针对加密和解密的统一密钥定义）。可采用相同步骤来以相反顺序完成加密和解密。

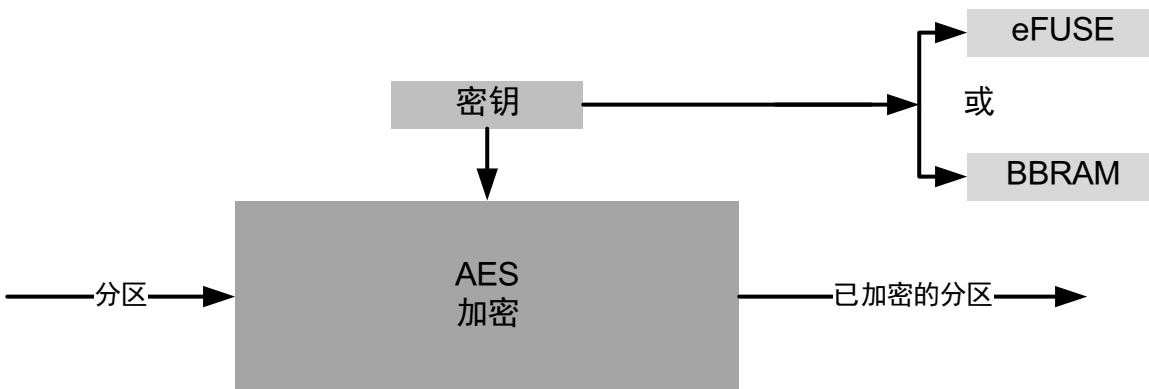
AES 属于迭代对称块密码，这表示它可以：

- 多次重复已定义的相同步骤
- 使用秘密密钥加密算法
- 在固定数量的字节上运行

加密进程

Bootgen 可以根据用户提供的加密命令和 BIF 文件中的属性来对启动镜像分区进行加密。AES 是一种对称密钥加密技术，它使用同一个密钥来进行加密和解密。使用启动镜像启动器件时，此器件上应包含用于对该启动镜像进行加密的密钥，并且此密钥应可用于解密流程。通常，此密钥存储在 eFUSE 或 BBRAM 内，创建启动镜像期间可通过 BIF 属性来选择密钥源，如下图所示。

图 12：加密进程图示

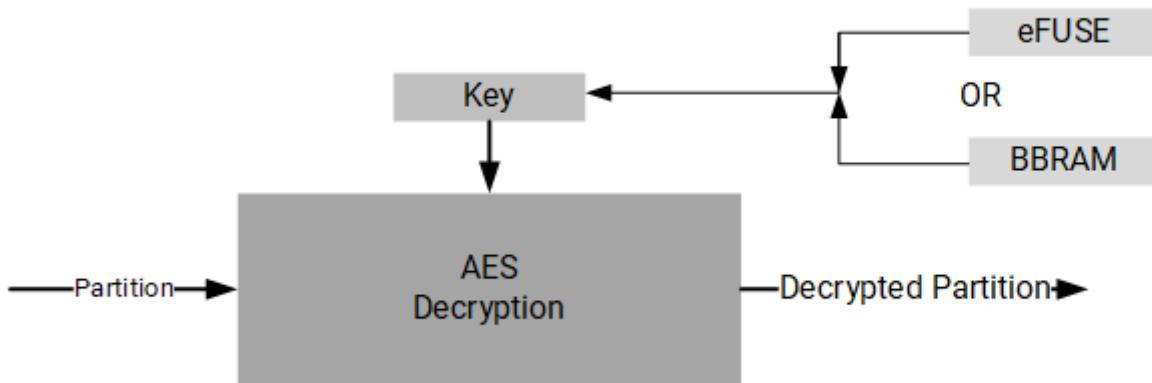


X21274-043022

解密进程

对于 SoC 器件，BootROM 和 FSBL 会在启动周期内对分区进行解密。BootROM 会从闪存读取 FSBL、执行解密、加载并交接控制权。FSBL 开始执行后，它会读取其余分区、执行解密然后加载这些分区。可从 eFUSE 或 BBRAM 检索对分区进行解密所需的 AES 密钥。通过读取启动镜像中的启动报头表的密钥源字段即可知晓加密密钥源。每个已加密的分区都会使用 AES 硬件引擎来进行解密。

图 13：解密流程图示



X21274-062320

对 Zynq 7000 器件分区进行加密

AMD Zynq™ 7000 SoC 器件使用嵌入式可编程逻辑 (PL) 散列消息认证码 (HMAC) 和含密码分组链接 (CBC) 模式的高级加密标准 (AES) 模块。

BIF 文件示例

为使用加密分区创建启动镜像，在 BIF 中使用 `aeskeyfile` 属性指定了 AES 密钥文件。为 BIF 文件中列出的要加密的每个镜像文件指定 `encryption=aes` 属性。BIF 文件 (`secure.bif`) 示例如下所示：

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

在命令行中，使用以下命令可生成含已加密的 `fsbl.elf` 和 `uboot.elf` 的启动镜像。

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

密钥生成

Bootgen 可生成 AES-CBC 密钥。Bootgen 使用 BIF 中指定的 AES 密钥文件来对分区进行加密。如果密钥文件为空或者不存在，Bootgen 会在 BIF 文件中指定的文件内生成密钥。如果在 BIF 中未指定密钥文件，并且针对任一分区已请求加密，那么 Bootgen 会在 BIF 所在目录内生成一个密钥文件，其文件名为 BIF 文件的名称加上 `.nky` 扩展名。以下是密钥文件样本。

```
Device          xc7z020c1g484  
Key 0          f878b838d8589818e868a828c8488808  
Key StartCBC   5C9D95ECBFEC8A1F12A8EB312362C596  
Key HMAC       00001111222233334444555566667777
```

对 Zynq MPSoC 器件分区进行加密

AMD Zynq™ UltraScale+™ MPSoC 使用 AES-GCM 核，其中包含 1 个 32 位基于字的数据接口，并支持 1 个 256 位密钥。AES-GCM 模式支持加密和解密、多个密钥源以及内置消息完整性检查。

运行密钥

良好的密钥管理方法包括最大限度减少秘密密钥或私钥的使用。这可通过使用 Bootgen 中启用的运行密钥选项来实现。

Bootgen 会创建经加密的安全报头，其中包含用户指定的运行密钥 (opt_key) 以及启用该功能时首个配置文件块所需的初始化矢量 (IV)。这样即可将器件上的 BBRAM 或 eFUSE 中存储的 AES 密钥的使用限制到 384 位内，从而显著降低其遭受旁路攻击的可能。opt_key 属性可用于指定运行密钥的使用方式。请参阅 [fsbl_config](#) 以获取有关作为 fsbl_config 属性的实参的 opt_key 值的更多信息。以下是 opt_key 属性的使用示例。

```
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  
  
}
```

AES 密钥 (.nky) 文件中提供的运行密钥名为 Key Opt，如以下示例所示。

图 14：运行密钥

Device	xczu9eg;
Key 0	9C42D9B74B633132F57C381D5CA4C7DF0829382CDBC455CDA08ECA62EB11D19D;
IV 0	42D3818AC135A365EDBD5316;
Key Opt	36AD8321ECA72E9F88E4F3A85ACDA27D1F50773E24B95067BA3BA75A3A62;

Bootgen 可生成加密密钥文件。随后会在 .nky 文件中生成运行密钥 opt_key，前提是在 BIF 文件中已启用 opt_key，如以上示例中所示。

要查看另一个运行密钥使用示例，请参阅 [使用运行密钥来保护开发环境中的器件密钥](#)。

如需了解有关该功能的更多详情，请参阅《Zynq UltraScale+ 器件技术参考手册》([UG1085](#)) 中的“密钥管理”部分。

密钥滚动

AES-GCM 还支持密钥滚动功能，其中整个加密镜像以较小的 AES 加密块/模块形式来展示。每个模块均使用其专用密钥进行加密。初始密钥存储在器件上的密钥源中，而每个后续模块的密钥则在前一个模块中进行加密（封装）。含滚动密钥的启动镜像可使用 Bootgen 生成。BIF 属性 `blocks` 可指定在创建多个较小的块以便加密时所采用的模式。

```
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  
}
```

注释：

- 密钥文件中的密钥数量始终与要加密的块数相等。
 - 如果密钥数量少于要加密的块数，那么 Bootgen 会返回错误。
 - 如果密钥数量多于要加密的块数，那么 Bootgen 会忽略（不读取）额外的密钥。
- 如要指定多个 Key/IV 对，应指定 `no. of blocks + 1` 对
 - 额外的 Key/IV 对将用于加密安全报头。
 - 单一 bif 中给定的任意 aes 密钥文件内不应存在重复的 Key/IV 对（Key0 和 IVO 除外）。

灰密钥/模糊密钥

用户密钥使用嵌入器件金属层的族密钥进行加密。此族密钥针对 AMD Zynq™ UltraScale+™ MPSoC 中的所有器件都相同。由此生成的结果被称为模糊密钥。模糊密钥可驻留在经过身份验证的启动报头中或 eFUSE 中。

```
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 创建镜像时执行以下操作：

1. 将来自 `bhiv.txt` 的 IV 置于启动报头中的 **BH IV** 字段中。
2. 将来自 `aes.nky` 的 IV0 置于启动报头中的“Secure Header IV”字段中。
3. 使用来自 `aes.nky` 的 Key0 和 IVO 对分区进行加密。

在 [附录 A: 用例与示例](#) 中提供了使用灰密钥/族密钥的另一个示例。

如需了解有关此功能的更多详情，请参阅《Zynq UltraScale+ 器件技术参考手册》([UG1085](#))。

密钥生成

Bootgen 具有生成 AES-GCM 密钥的功能。它使用经 NIST 核准的计数器模式 KDF，并使用 CMAC 作为虚拟随机函数。Bootgen 可使用种子作为输入，以防用户因密钥滚动而需要从种子衍生多个密钥。如果指定种子，则将使用种子衍生出密钥。如果未指定种子，则将基于 Key0 来衍生密钥。如果指定的密钥文件为空，那么 Bootgen 会使用基于时间的随机化（非 KDF）来生成种子，并将其作为输入以供 KDF 用于生成其他 Key/IV 对。

注释：

- 如果指定一个密钥文件并生成其他密钥文件，那么 Bootgen 可确保将首个分区的加密文件内的 Key0/IV0 对用于所生成的密钥。例如，对于完整启动镜像，第一个分区是启动加载程序。
- 如果针对首个分区生成一个加密文件并使用为后续某一分区指定的 Key0/IV0 来生成其他加密文件，则 Bootgen 将退出并返回错误，声明使用的 Key0/IV0 对不正确。

密钥生成

以下显示了密钥文件样本。

图 15：密钥文件样本

Device	xczu9eg;
Key 0	AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
IV 0	11198912D243EF0AFEAC8970;
Key 1	C023E238AC903111DEF0AABB98C1CCDDDEFF021001289011198C1E238AC34012;
IV 1	111DEF0AABBCCDDEEFF00112;
Key 2	11456A9B8764DE111444C023E238A98C1CCC9031177112E01289011198CFF010;
IV 2	9C64778CBAF48D6DDE13749B;
Key Opt	229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;

模糊密钥生成

Bootgen 可使用族密钥和用户提供的 IV 来对红密钥进行加密，以生成模糊密钥。族密钥由 AMD 安全小组提供。如需了解更多信息，请参阅 [familykey](#)。为生成模糊密钥，Bootgen 会从 BIF 文件提取以下输入。

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

用于生成模糊密钥的命令为：

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

黑密钥/PUF 密钥

黑密钥存储解决方案使用加密强度极高的密钥加密密钥 (KEK)（从 PUF 生成）来加密用户密钥。随后，生成的黑密钥可存储在 eFUSE 中或者包含在经过身份验证的启动报头内。

```
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  
}
```

要查看另一个黑密钥使用示例，请参阅 [附录 A: 用例与示例](#)。

多个加密密钥文件

先前版本的 Bootgen 支持通过使用单一加密密钥对多个分区进行加密来创建启动镜像。针对每个分区重复使用相同密钥。这是一个安全漏洞，不推荐使用此方法。流程中每个密钥应仅使用一次。

Bootgen 支持针对每个分区使用独立的加密密钥。如果存在多个密钥文件，请确保每个加密密钥文件都使用相同的 Key0（器件密钥）、IV0 和运行密钥。如果每个加密密钥文件中上述密钥不同，那么 Bootgen 将不允许创建启动镜像。您必须指定多个加密密钥文件，针对镜像中每个分区指定一个密钥文件。使用针对分区指定的密钥来对分区进行加密。

注释：由于存在多个可加载节，您可为每个分区创建专用密钥文件，方法是在分区专用的密钥文件目录内给密钥文件名追加 .1、.2、.n 等。

以下代码片段显示了 1 个加密密钥文件样本：

```
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
}
```

- `fsbla53.elf` 分区是使用来自 `key_p0.nky` 文件的密钥进行加密的。
- 假定 `hello.elf` 具有 3 个分区，因为它具有 3 个可加载节，而 `hello.elf.0` 分区是使用来自 `test2.nky` 文件的密钥加密的。
- 此外，`hello.elf.1` 分区是使用来自 `test2.1.nky` 的密钥加密的。
- `hello.elf.2` 分区是使用来自 `test2.2.nky` 的密钥加密的。

对 Versal 器件分区进行加密

AMD Versal™ 器件使用 AES-GCM 核，该核支持 256 位密钥。创建安全镜像时，可选择对启动镜像中的每个分区进行加密。密钥源和 aes 密钥文件对于加密都是必需的。

注释：对于 Versal 自适应 SoC，启用加密时，必须指定每个分区的 AES 密钥文件和密钥源。根据使用的密钥源，在每个指定的 aes 密钥文件内应使用相同的 Key0，反之亦然。

密钥管理

良好的密钥管理方法包括最大限度减少秘密密钥或私钥的使用。这可通过在启动镜像内的不同分区间使用不同 Key/IV 对来实现。这样即可将器件上的 BBRAM 或 eFUSE 中存储的 AES 密钥的使用限制到 384 位内，从而显著降低其遭受旁路攻击的可能。

```
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}
}
}
```

密钥滚动

AES-GCM 还支持密钥滚动功能，其中整个加密镜像以较小的 AES 加密块/模块形式来展示。每个模块均使用其专用密钥进行加密。初始密钥存储在器件上的密钥源中，而每个后续模块的密钥则在前一个模块中进行加密（封装）。您可使用 Bootgen，通过密钥滚动来生成启动镜像。BIF 属性 blocks 可指定在创建多个较小的块以便加密时所采用的模式。

注释：对于 Versal 自适应 SoC，默认密钥滚动以 32 KB 数据为单位来执行。您通过 blocks 属性所选的密钥滚动将应用于每个 32 KB 区块内。这是对所使用的散列方法的补充。如果启用 DPA 密钥滚动对应措施，则会影响启动时间。请参阅启动时间估算其电子数据表以了解计算方式。

```
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 = bbram_red_key,
            aeskeyfile = key1.nky,
            dpacm_enable,
```

```
        blocks = 8192(20);4096(*),
        file = lpd_data.cdo
    }
partition
{
    id = 0x0B, core = psm,
    encryption = aes,
    keysrc = bbram_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
}
}
}
```

注释：

- 密钥文件中的密钥数量始终与要加密的块数相等。
- 如果密钥数量少于要加密的块数，那么 Bootgen 会返回错误。
- 如果密钥数量多于要加密的块数，那么 Bootgen 会忽略额外的密钥。

密钥生成

Bootgen 可生成 AES-GCM 密钥。它使用经 NIST 核准的计数器模式 KDF，并使用 CMAC 作为虚拟随机函数。Bootgen 可使用种子作为输入，以防因密钥滚动而需要从种子衍生多个密钥。如果指定种子，则将使用种子衍生出密钥。如果未指定种子，则将基于 Key0 来衍生密钥。如果指定的密钥文件为空，那么 Bootgen 会使用基于时间的随机化（非 KDF）来生成种子，并将其作为输入以供 KDF 用于生成其他 Key/IV 对。适用下列条件。

- 如果指定一个密钥文件并生成其他密钥文件，那么 Bootgen 可确保将首个分区的加密文件内的 Key0/IV0 对用于所生成的密钥。
- 如果针对首个分区生成一个加密文件并使用为后续某一分区指定的 Key0/IV0 来生成其他加密文件，则 Bootgen 将退出并返回错误，声明使用的 Key0/IV0 对不正确。
- 如果未指定密钥文件并为分区选择启用加密，那么默认情况下 Bootgen 会生成 aes 密钥文件，其名称与分区名称相同。这样 Bootgen 即可确保针对每个分区使用不同的 aeskeyfile。
- Bootgen 支持针对因存在多个可加载节而创建的每个分区使用专用密钥文件，方法是读取/生成密钥文件名，并在分区专用的密钥文件目录内对密钥文件名追加“1”、“2”…“n”。

黑密钥/PUF 密钥

黑密钥存储解决方案使用加密强度极高的密钥加密密钥 (KEK)（从 PUF 生成）来加密用户密钥。随后，生成的黑密钥可存储在 eFUSE 中或者包含在经过身份验证的启动报头内。示例：

```
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 报头加密

对于 Versal 自适应 SoC，Bootgen 会在“metaheader”属性下明确提及加密时对 Meta 报头进行加密。可在 bif 中使用“metaheader”下的参数指定所使用的 aeskeyfile。用法片段如下所示。

注释：Meta 报头加密包括除“镜像报头表”之外的所有报头。

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

适用下列条件。

- 如果针对 Meta 报头未指定具体 aeskeyfile，那么 Bootgen 会生成名为 meta_header.nky 的文件，并在加密期间使用此文件。
- 如果在 bif 中存在启动加载程序，那么必须加密启动加载程序才能加密 Meta 报头。对于部分 PDI，是否加密 Meta 报头为可选操作。
- 对 Meta 报头进行加密时，部分 PDI 可作为经过身份验证的额外数据来添加，用于确保镜像报头表正确无误。

使用身份验证

AES 加密属于含对称密钥的自验证算法，即要加密的密钥与要解密的密钥相同。此密钥必须加以保护，因为它属于秘密密钥（因此存储至内部密钥空间内）。存在另一种身份验证格式，即 Rivest-Shamir-Adleman (RSA) 格式。RSA 属非对称算法，即要验证的密钥与用于签名的密钥不同。要进行身份验证，需一对密钥。

- 使用秘密密钥/私钥来执行签名

- 验证是使用公钥完成的

此公钥无需受保护，并且无需特殊安全存储空间。这种形式的身份验证可配合加密一起用于确保真实性和保密性。RSA 可配合已加密分区或未加密分区一起使用。

RSA 不仅拥有使用公钥的优势，而且还具有先验证后解密的优势。RSA 公钥的散列必须存储在 eFUSE 中。AMD SoC 器件支持先对分区数据进行身份验证，然后再将其发送到 AES 解密引擎。此方法可用于帮助预防对解密引擎本身发起的攻击，因为它可确保在执行任何解密操作前，分区数据已经过验证。

在 AMD SoC 中，使用两对公钥和秘密密钥 - 主公钥和秘密密钥以及辅助公钥和辅助秘密密钥。主公钥/秘密密钥对的功能是用于对辅助公钥/秘密密钥对进行身份验证。辅助密钥的功能是签署/验证分区。

用于描述密钥的首字母缩略词的第一个字母为 P（表示主 (primary) 密钥）或 S（表示辅助 (secondary) 密钥）。用于描述密钥的首字母缩略词的第二个字母为 P（表示公钥 (public)）或 S（表示秘密密钥 (secret)）。存在 4 种可能的密钥：

- PPK = 主公钥
- PSK = 主秘密密钥
- SPK = 辅助公钥
- SSK = 辅助秘密密钥

Bootgen 可通过 2 种方式创建身份验证证书：

- 提供 PSK 和 SSK。SPK 签名使用这 2 项输入来即时计算。
- 提供 PPK 和 SSK，并提供 SPK 签名作为输入。在 PSK 未知的情况下，可使用此方法。

主密钥采用散列化并存储在 eFUSE 中。此散列将与 FSBL 存储在启动镜像中的主密钥的散列进行比对。可使用随 Vitis 提供的独立驱动将此散列写入 PS eFUSE 存储器。

以下是 BIF 文件示例：

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

如需了解特定于器件的身份验证信息，请参阅：

- [Zynq 7000 身份验证证书](#)
- [Zynq UltraScale+ MPSoC 身份验证证书](#)
- [Versal 自适应 SoC 身份验证证书](#)

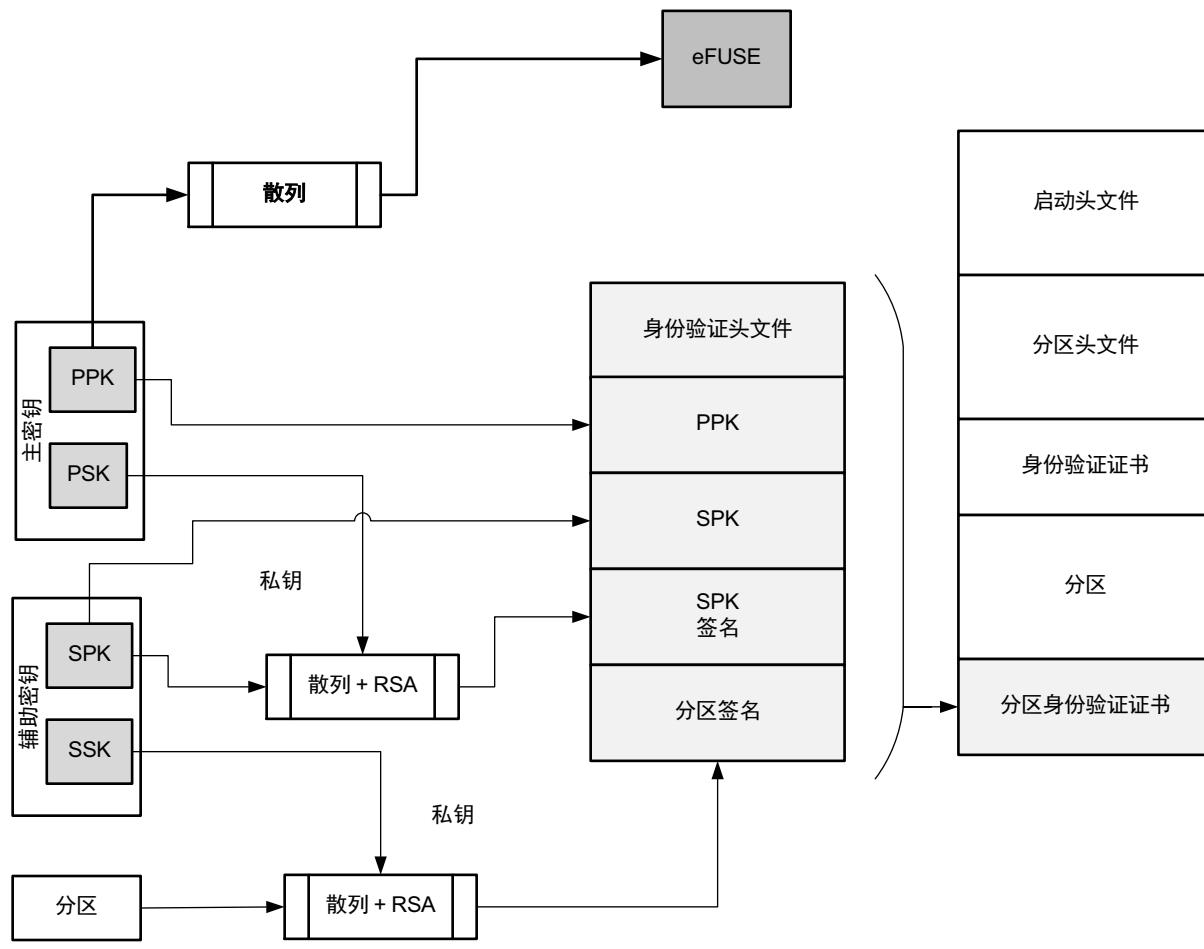
签名

下图显示了分区的 RSA 签名。Bootgen 通过安全的工具来使用秘密密钥对分区进行签名。签名过程如以下步骤所述：

1. PPK 和 SPK 存储在身份验证证书 (AC) 中。
2. SPK 使用 PSK 来签名，以获取 SPK 签名；并且此签名同样存储在 AC 中。
3. 分区使用 SSK 来签名，以获取分区签名，此签名填充到 AC 中。

4. AC 将附加到选中执行身份验证的每个分区之前或之后（视器件而定）。
5. PPK 将散列化并存储在 eFUSE 中。

图 16：RSA 分区签名



下表显示了身份验证选项。

表 35：身份验证密钥支持的文件格式

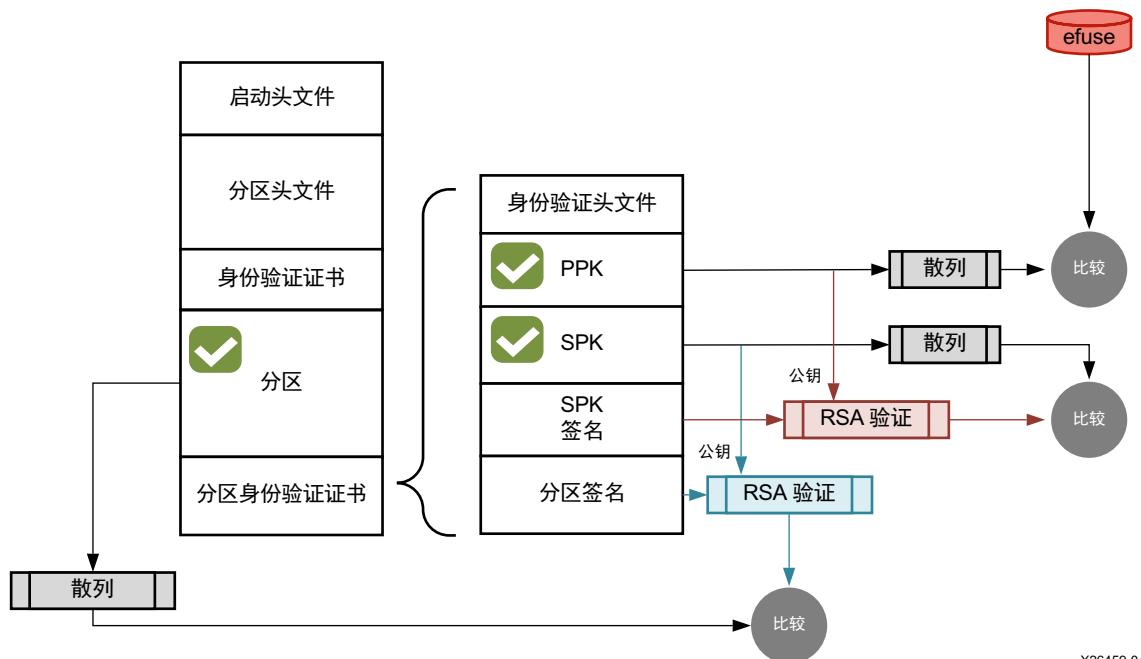
键	名称	描述	支持的文件格式
PPK	Primary Public Key	该密钥用于对分区进行身份验证。 对分区进行身份验证时应始终指定该密钥。	*.pem *.pub
PSK	Primary Secret Key	该密钥用于对分区进行身份验证。 对分区进行身份验证时应始终指定该密钥。	*.pem
SPK	Secondary Public Key	可指定该密钥用于对分区进行身份验证。	*.pem *.pub
SSK	Secondary Secret Key	可指定该密钥用于对分区进行身份验证。	*.pem pub

验证

在器件中，BootROM 可验证 FSBL，而 FSBL 或 U-Boot 则会使用公钥来验证后续分区。

1. 验证 PPK：此步骤用于确定主密钥的真实性，主密钥用于对辅助密钥进行身份验证。
 - a. PPK 可从启动镜像的 AC 中读取
 - b. 生成 PPK 散列
 - c. 散列化 PPK 将与从 eFUSE 检索得到的 PPK 进行比对
 - d. 如果两者相同，则主密钥可信，否则安全启动失败
2. 验证辅助密钥：此步骤用于确定辅助密钥的真实性，辅助密钥用于对分区进行身份验证。
 - a. SPK 可从启动镜像的 AC 中读取
 - b. 生成 SPK 散列
 - c. 使用 PPK 通过验证 AC 中存储的 SPK 签名来获取 SPK 散列
 - d. 将步骤 (b) 和步骤 (c) 中的散列进行比对
 - e. 如果两者相同，则辅助密钥可信，否则安全启动失败。
3. 验证分区：此步骤用于确定要启动的分区的真实性。
 - a. 分区可从启动镜像读取。
 - b. 生成分区散列。
 - c. 使用 SPK 通过验证 AC 中存储的分区签名来获取分区散列。
 - d. 将步骤 (b) 和步骤 (c) 中的散列进行比对
 - e. 如果两者相同，则分区可信，否则安全启动失败

图 17：验证流程图示



Bootgen 可通过 2 种方式创建身份验证证书：

- 提供 PSK 和 SSK。SPK 签名使用这 2 项输入来即时计算。
- 提供 PPK 和 SSK，并提供 SPK 签名作为输入。在 PSK 未知的情况下，可使用此方法。

Zynq UltraScale+ MPSoC 身份验证支持

AMD Zynq™ UltraScale+™ MPSoC 使用 RSA-4096 身份验证，这表示主密钥和辅助密钥大小均为 4096 位。

NIST SHA-3 支持

注释：对于 SHA-3 身份验证，请始终使用 Keccak SHA-3 来计算启动报头、PPK 散列和启动镜像上的散列。NIST-SHA3 适用于 ROM 未加载的所有其他分区。

生成的签名使用基于下表的 Keccak-SHA3 或 NIST-SHA3：

表 36：身份验证签名

所用身份验证证书 (AC)	签名	SHA 算法和 SPK eFUSE	用于生成签名的秘密密钥
分区报头 AC (由 FSBL/FW 加载)	SPK 签名	如果使用 SPKID eFUSE，则使用 Keccak；如果使用用户 eFUSE，则使用 NIST	PSK
	BH 签名	始终使用 Keccak	SSK _{header}
	报头签名	始终使用 Nist	SSK _{header}
BootLoader (FSBL) AC (由 ROM 加载)	SPK 签名	始终使用 Keccak；针对 SPK 始终使用 SPKID eFUSE	PSK
	BH 签名	始终使用 Keccak	SSK _{Bootloader}
	FSBL 签名	始终使用 Keccak	SSK _{Bootloader}
其他分区 AC (由 FSBL FW 加载)	SPK 签名	如果使用 SPKID eFUSE，则使用 Keccak；如果使用用户 eFUSE，则使用 NIST	PSK
	BH 签名	始终使用 Keccak 填充	SSK _{Partition}
	分区签名	始终使用 NIST 填充	SSK _{Partition}

示例

示例 1：以下 BIF 文件利用一组密钥文件对分区进行身份验证：

```
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  
}
```

示例 2：以下 BIF 文件针对每个分区使用一个独立辅助密钥对分区进行身份验证：

```
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

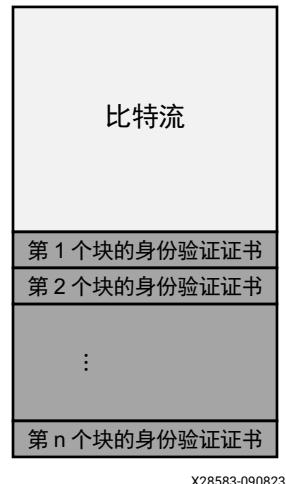
    // ATF (Partition-1)
    [
        destination_cpu = a53-0,
        authentication = rsa,
        exception_level = el-3,
        trustzone = secure,
        spk_id = 0x02,
        sskfile = secondary_p2.pem
    ] b131.elf

    // UBOOT (Partition-2)
    [
        destination_cpu = a53-0,
        authentication = rsa,
        exception_level = el-2,
        spk_id = 0x03,
        sskfile = secondary_p3.pem
    ] u-boot.elf
}
```

使用外部存储器执行比特流身份验证

比特流的身份验证不同于其他分区。FSBL 可完整包含在 OCM 内，从而在器件内部进行身份验证和解密。对于比特流，由于文件大小过大，无法完整包含在器件内，必须使用外部存储器。使用外部存储器会造成难以维持安全性，因为攻击者可能有权访问此外部存储器。请求对比特流进行身份验证时，Bootgen 会将整个比特流分割为多个 8 MB 块，并为每个块提供身份验证证书。如果比特流大小并非 8 MB 的倍数，那么最后一个块包含剩余比特流数据。同时启用身份验证和加密时，首先在比特流上执行加密，然后 Bootgen 会将已加密的数据分割为多个块，并为每个块提供身份验证证书。

图 18：使用外部存储器执行比特流身份验证



含增强型 RSA 密钥撤销的用户 eFUSE 支持

增强型 RSA 密钥撤销支持

RSA 密钥支持在不撤销所有分区的辅助密钥的前提下撤销某一分区的辅助密钥。

注释：所有分区的主密钥都应相同。

这是使用含 BIF 参数 `spk_select` 的 USER_FUSE0 到 USER_FUSE7 eFUSE 实现的。

注释：最多可撤销 256 个密钥（全部无需使用）。

以下 BIF 文件样本显示了增强型用户 eFUSE 撤销。镜像报头与 FSBL 通过以下 BIF 输入来使用不同 SSK 来进行身份验证（分别使用 `ssk1.pem` 和 `ssk2.pem`）。

```
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` 指示针对该分区使用 `spk_id` eFUSE。
- `spk_select = user-efuse` 指示针对该分区使用用户 eFUSE。

由 CSU ROM 加载的分区始终使用 `spk_efuse`。

注释：`spk_id` eFUSE 可指定有效的密钥。因此，ROM 会根据 SPK ID 检查 `spk_id` eFUSE 的整个字段，以确保其位对位匹配。

用户 eFUSE 可指定无效（已撤销）的密钥 ID。因此，固件（非 ROM）会检查表示 SPK ID 的给定用户 eFUSE 是否已烧录。

注释：在以上示例中，FSBL 与 application2 使用相同的 `spk_id`。但这两个密钥可分别调用，因为其中之一是基于 SPK_ID eFUSE 来进行检查的，另一个则是基于 User eFUSE 来进行检查的。

密钥生成

Bootgen 支持生成 RSA 密钥。或者，您可使用外部工具（如 OpenSSL）来创建密钥。Bootgen 可在 BIF 文件指定的路径中创建密钥。

下图显示了 RSA 私钥文件样本。

图 19：RSA 私钥文件样本

```
-----BEGIN RSA PRIVATE KEY-----
MIIJKAIKBAAKCgEA4ppimme6TvPT5+JB2CgXQLU9AyStbnEr21EJu+ZpR9HZ5Plq
6KbOcFuV6q3EKvI5PJsMS0yHpvf/11/uTPxvUT6Im5goMyaskz0PS3xTWuYoSDba
YD5021Pi5xBrsWvys6YcIbLTbk2+c86o0Rr/sdQtLR0pbsLfuBFcKMEsK19N12k
E116DM1Tjh9KSpZOzmj7yew2Rm857QqQp8su1Vi4qdtIr58+MoQxeETeHcN+zq4
drlUsUqX3msVb9z0rRwYrBVsksWr5d+xj+cAUpiPjeMGRXg00L6gEGGPTjnqQtG
YFCoCFcBL4JknHF/yMyV7f6wh2xtkKbme+Kuovcz/pQVKEGELkQ9kjweBf5c8Vm
b13NvkrAUOKYLM+py0uY/PGjtz685W964LocrT+TRROi4FGotYzk2XmJtODO5dyH
Lw58IOT3zAYwaC/98bUDGYP6kJ9+YqprerLm2U55EW30PPodjHYihLmbj1pvmu4g
oZ9tXJPch/uRk/tv3e53P2JhWKwdB72FU18HegSkCWAffJwCVFwATettzGlhtz+
Ww3eBAQi9fBgr6YERwKOOLopaRQizPaC/8XG8u0bTE3MdvsJK/IIOAqVnT17Dfs
QKzTzap8+Iwx/vuaWaiLd0qYCDKKMlGGz5bQhegRnk0I/lpOK1lPRL8wH0CAwEA
AQKCAgA3qhscu0XgZq8gYEkyey67G4pgUks0PSK7n3qXqNM17FvtToO/oPJHUYgz
PPpaXmRHCGNsH+GWchM08gDU8pKWeJkQN8FwR0jPZolyTpkfVDiC/M6KI+luEZ9E
i2kbQgNb+4Ig6kvYzO2/gR2za6Rn0shli3q4F4mMykVYX5NQXmI/Doa2ph1AndQX
r0IObnvvYoSvppHynXIKU7UTMutPR1sdhpuInouWErzJbPOimrAzofU3FA7Y
eU+righk2ekJpL3TKTzqZ3mh85ABF0YrQfPtWZ1/6A0nInF6apclxHgGQKn2WoEV
DZ/vekYcqnoOGK1+qtkDVqx5tEaX1XG1c0PBWg5aofkpNZ0K0wOG6iCueNvATcJ9
RoMq7c7zZOYh4SzWgSjP3a8neGcnhG0T6BGYCGjPXRW2Y6ri/7lrDcOBVSc3zS8p
IVKAkBp13PIgZ1hMnxdc60RPh8dhXR0TuA3+1SyGx37Ad9260UeHHJIpz28DktTg
CY7RU5SDSh6wDuDbheliu4nzZDGWeKq9zeAxzGzhIn0zcxpWvG54uHTHNnqBEFJ2S
ZSJ8sq4aYiZCiW/PrqKgg8wBygKcEtr3/LcAm4r3p19mHk1555QQNdpk+ba+3GLp
bEy0889KwCyPKfWY5p16VNgyLycxe/TofMDCHQARA2wLrnN1sQQKCAQEAA80nn83su
OYN90c22owfm/MHGJ6mFi5LpRtgylWbcAbDsZs7rjQ4Iz46JQ1Mp10IpNbVub7
sWOFUX7sVo0XZMS1+Eps2Dq021+7hY6+MGALtPpg9n2Z91fCyVxfNqv5SiMv6Te
6/jur69KiwhztYf17JK4GGUdcCwyAwMTdgm3pQDDH99Vp436klv41MyjeQaIp0/
FzkiklfYn84j9jvtagoMk0fzaickiOGss4ci0ds3DEgGC9x1hDK1s9UFpk1Pfw=7
qYnsT7XIwoTCBrvQ11Kp5fLZUhsRsIQV82u44IPfcU3xWgeyInSGx0RfSV5RWoV
v9sJPVsFlxE5EQKCAQEAA7nFNK5gbPKA0nxKTEm1ZMhp99/YqRxpj5irmXmrF54cn
sZPpG/dvbJBXILAd9hsSYjw8FNY5ehJhL9IqzEVavFr8SAvu2FyI9MN0d9wUvpJG
55JxK9K090uSzaXZVimV/5xumbnnywzZwgxs1sAYoNy+8sov1Z1KQxZzeUaoham
VVuL1HdRzE0afrcFsnfugID172Mb14t2cKTFtek/iYAvF9bk076upkPmMu4V7yFT
of9QFkq8qBRthEpvaKNTObpU5TrzsxUH3rYXvnAzgpEXEJdeVVFYzSLf4SC45mx
Gfp37pYetPBKVrUesuEvQ70IeoicGRXFqC9TPmYwrQKCAQEAA5D1CoPbAD+7ejVsD
a4FFX2K+7rin86A4v1q9hlzAK8n6jhyzeRpgh7jgFiaj9hRnppVx3pdSD+6DJCh
UTV+a+fcuMnBVGqK/3+ZYhvfK2z/rqJyUuzFXDxWYROANz7GY5seKDCZfhGEg0dV
DIg6XV5sGvsuQJyj+HE0xo8dPlCxe9fyNrWEgvQkzxgX64gX1mvXZPbs//F3EIne
6801kyz3d1LEJ2wJ3V2pdc0BnvE4175K1/f9zCTgDtKe6m7/Q0qu0MreyJf/HWyy
UmLP0BdlAogfdIkApR0rKvym7mlGQUMWXaq8sTS1FpPxWYI4TpFwi2aXAg2a9w3
qdKVsQKCAQBH8nolcFT/mxusBY91kDSRvPB06qse8UPC3zNmowyy25nv8jd/opp
iLgxjdLMkuieJ7ajluwq8GbQ5iL2cEftrs8yR9L/SG0HceSoQjKDZzAuHD0IVNuAS
CoS2dse4nv26zjn1Os2BvmHvvu1/BvtJFrKrUs8MT/KZ3jabD6nbEkhGX+m25c
JhvLhnA6pMOb1M1MzWu/8vH/FVCoEqxwUfrjzhy6BlRuqOhWIacOq9CvffltcImy
cc+F7mvld/rB3X6GWJ52N+9S/UDXFxF2wA9q171gYE5DL/fD1+bh7GI+fK8VCH2
2P0lbCtiMF5oxVu28fdx9r7TcxhdL2VAoIBADmGYfxvgEqhALqdW2QmtRRNisWQ
y0/RfED7dNtN8o5vjBCbrOV/tQ3Ddb7a0kw01NFrlxR7K1ki98SkKN0EiCrpRfc
+ccs6kAST2cPH/nGG91br0Am9FOG2q5cX6kDk1hqHe+1UYm/34a+2wN0/CwAh7MH
gECABtqx9GCD/DJ1+n5ocrYk5RsQJrtnwP4L8X24dRiMiRMIsS4V9uyyRLQTWV/
k3TOjRgL5eRKbcVw7c8kmaGDWFM/eVLLQW+wEa0wY+TdSUhlyvgsG5yijkhCAEe
/+Az0w5ZulvnLbj5eXK1ULWIS1osDCBfJepuINHoUpBwsGzFb7zXtpK2X1M=
-----END RSA PRIVATE KEY-----
```

注释：公用部分通常以扩展名 .pub 来引用。这可从包含公用部分和专用部分的私钥中提取。私钥通常采用扩展名 .pem。要生成公钥部分，请使用 ppkfile/spkfile 替代以上示例中的 pskfile/sskfile。

BIF 示例

BIF 文件样本 generate_pem.bif 如下所示：

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

命令

用于生成密钥的命令如下所示：

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

适用于 eFUSE 的 PPK 散列

Bootgen 可生成 PPK 散列并将其存储在 eFUSE 中以使 PPK 可信任。仅限非对称硬件信任根 (AHWRoT) 须执行此步骤，AMD Zynq™ UltraScale+™ MPSoC 器件的 AHWRoT 可跳过此步骤。来自 efuseppksha.txt 的值可烧录到 eFUSE 中以供 AHWRoT 使用。

如需了解有关 BBRAM 和 eFUSE 烧录的更多信息，请参阅《BBRAM 和 eFUSE 烧录》(XAPP1319)。

BIF 文件示例

以下是一个简单的 BIF 文件：generate_hash_ppk.bif。

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

命令

用于生成 PPK 散列以进行 eFUSE 烧录的命令如下所示：

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

Versal 身份验证支持

Bootgen 支持使用 RSA-4096、ECDSA P384 曲线和 P521 曲线来执行 Versal 自适应 SoC 身份验证。NIST SHA-3 用于计算所有分区/报头上的散列。散列上计算所得签名置于 PDI 中。

注释：不同于 Zynq 器件和 Zynq UltraScale+ MPSoC，对于 Versal 自适应 SoC，身份验证证书置于分区之前。不支持使用 ECDSA P521 曲线对启动加载程序分区 (PLM) 进行身份验证，因为 BootROM 仅支持 RSA-4096 或 ECDSA-P384 身份验证。但是，可使用 P521 对任何其他分区进行身份验证。

Meta 报头身份验证

对于 Versal 自适应 SoC，Bootgen 可根据 bif 属性 “metaheader” 下的参数对 Meta 报头进行身份验证。用法片段如下所示。

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

身份验证优化

从 v2024.1 版开始，对于 Versal 自适应 SoC，您可以选择缩短身份验证时间。

如果您对 PLM 加载的 Meta 报头和分区使用相同的 SPK（具有相同的身份验证算法和密钥强度），就有可能避免 SPK 和分区数据签名确认。您可以启用使用存储在可选镜像报头表中的身份验证分区摘要。分区摘要是 SHA 摘要，用于在启动时确认分区数据的完整性。

读取 Meta 报头时，PLM 会进行身份验证并复制分区摘要表，以便在分区加载时使用。对于表中有摘要条目的任何分区，均可跳过身份验证。在此情况下，分区数据会直通 SHA 引擎，得到的摘要与镜像报头表中为该分区存储的已完成身份验证的摘要值进行比对。

如需了解如何启用身份验证优化，请参阅 [enable_auth_opt](#)。

适用于 eFUSE 的 PPK 散列

Bootgen 可生成 PPK 散列并将其存储在 eFUSE 中以使 PPK 可信任。仅限含 eFUSE 模式身份验证才需执行此步骤，针对 AHWRoT 可跳过此步骤。来自 `efuseppksha.txt` 的值可烧录到 eFUSE 中，以便使用 eFUSE 来为 AHWRoT 执行身份验证。

BIF 文件示例

以下是一个简单的 BIF 文件：`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}  
    }  
}
```

命令

用于生成 PPK 散列以进行 eFUSE 烧录的命令如下所示：

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

用于 Versal 自适应 SoC 的累积安全启动操作

表 37：累积安全启动操作

启动类型	操作			硬件加密引擎
	身份验证	解密	完整性（校验和验证）	
非安全启动	否	否	否	无
非对称硬件信任根 (A-HWRoT)	是（必需）	否	否	RSA/ECDSA（含 SHA3）
对称硬件信任根 (S-HWRoT)（使用 eFUSE 黑密钥强制解密 PDI）	否	是（必需的 PLM 和 Meta 头文件应使用 eFUSE KEK 来加密）	否	AES-GCM

表 37：累积安全启动操作 (续)

启动类型	操作			硬件加密引擎
	身份验证	解密	完整性（校验和验证）	
A-HWRoT + S-HWRoT	是（必需）	是（必需）	否	RSA/ECDSA（含 SHA3 和 AES-GCM）
对 PDI 进行身份验证 + 解密	是	是（密钥源可为 BBRAM 或 eFUSE）	否	RSA/ECDSA（含 SHA3 和 AES-GCM）
解密（使用用户所选密钥。密钥源可为以下任意类型，如 BBRAM/BHDR，甚至是 eFUSE）	否	是	否	AES-GCM
校验和验证	否	否	是	SHA3

Versal 散列方案

AMD Versal™ 自适应 SoC 器件引入了全新散列方案，旨在执行分区身份验证时尽可能缩短 PLM 的启动时间并减小缓冲器空间。该散列方案主要核心是在当前数据块中包含下一个数据块的散列（类似于密钥滚动）。这样即可将单一签名用于整个分区而不必考量分区大小，同时消除了 PLM 本身内部的缓冲器散列需求。此方案用于除启动加载程序外的所有分区。此数据块每次均散列化，它被称为安全区块。对于 Versal，此区块大小为 32 KB。

该散列方案如下表所示：

表 38：分区区块划分方案

分区区块计数	分区区块划分方案	注释
CHUNK 0	[Authentication Certificate - Partition Sign Field + SECURE HEADER + GCM TAG + SECURE_CHUNK_SIZE + HASH OF CHUNK 1]	此数据会加以散列化，随后签名。此签名置于 AC 的“Partition Signature”（分区签名）字段内
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]	

适用于 AMD Versal™ 的 SECURE_CHUNK_SIZE 为 32 KB。

注释：对于加密用例，这样可使用户密钥滚动完全包含在散列区块中。

使用 HSM 模式

在当前加密技术中，所有算法均公开，因此保护私钥/秘密密钥就显得至关重要。硬件安全模块 (HSM) 是专用加密处理器件，专为保护加密密钥生命周期而设计。因为此模块仅将公钥传递给 Bootgen，而不会传递私钥/秘密密钥，所有可提升密钥处理安全性。

在某些企业中，由信息安全部负责安全的嵌入式产品的量产发布。信息安全部可使用 HSM 执行数字签名，并使用独立安全服务器执行加密。HSM 和安全服务器通常驻留在安全区域内。HSM 是安全密钥/签名生成器件，可生成私钥、使用该私钥对分区进行签名，并将 RSA 密钥的公用部分提供给 Bootgen。私钥仅驻留在 HSM 内。

HSM 模式下的 Bootgen 仅使用公钥和由 HSM 创建的签名来生成启动镜像。HSM 接受由 Bootgen 生成的分区散列值，并根据散列和秘密密钥返回签名块。

相比于 HSM 模式，标准模式下的 Bootgen 使用 AES 加密密钥和通过 BIF 文件提供的秘密密钥来分别对镜像中的分区进行加密和身份验证。输出为经过加密和身份验证的单一启动镜像。对于身份验证，您必须提供公钥 (public key) 和私钥 (private key) /秘密密钥 (secret key) 的集合。私钥/秘密密钥供 Bootgen 用于签署分区并创建签名。这些签名与公钥一起嵌入最终启动镜像。

如需了解有关适用于 FPGA 的 HSM 模式的更多信息，请参阅 [HSM 模式](#)。

使用高级密钥管理选项

与私钥相关联的公钥为 `ppk.pub` 和 `spk.pub`。HSM 接受由 Bootgen 生成的分区散列值，并根据散列和秘密密钥返回签名块。

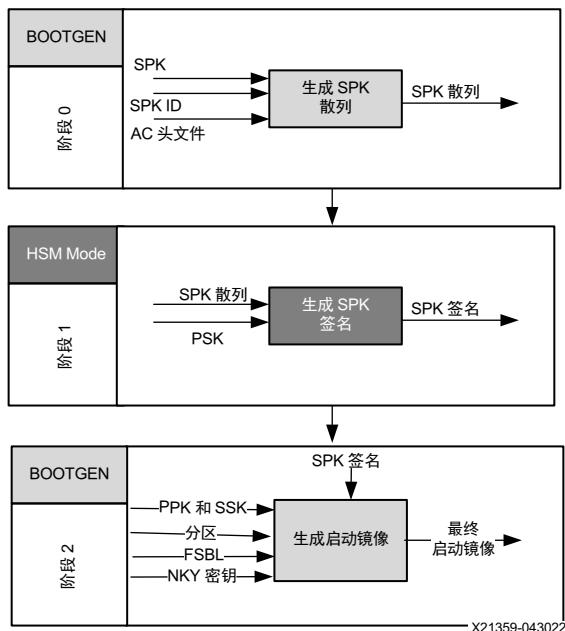
注释：不支持采用 HSM 流程来执行 mcs 格式启动镜像生成。

使用 HSM 模式创建启动镜像：PSK 未共享

下图显示了使用 HSM 模式时的阶段 0 到阶段 2 启动栈。它通过分发 SSK 来减少步骤数量。

此图使用 AMD Zynq™ UltraScale+™ MPSoC 来演示各阶段。

图 20：通用 3 阶段启动镜像



启动流程

使用 HSM 模式创建启动镜像与使用标准流程利用以下 BIF 文件创建启动镜像类似。

```
all:
{
    [auth_params] ppk_select=1;spk_id=0x8
    [keysrc_encryption] bbram_red_key
    [pskfile] primary.pem
    [sskfile] secondary.pem
    [
        bootloader,
        encryption=aes,
        aeskeyfile=aes.nky,
        authentication=rsa
    ] fsbl.elf
    [destination_cpu=a53-0,authentication=rsa] hello_a53_0_64.elf
}
```

使用 HSM 模式创建启动镜像

阶段 0：为 SPK 生成散列

可信人员使用主秘密密钥创建 SPK 签名。SPK 签名位于身份验证证书报头、SPK 和 SPK ID 上。要为上述对象生成散列，请使用以下 BIF 文件代码片段。

```
stage 0:
{
    [auth_params] ppk_select=1;spk_id=0x3
    [spkfile] keys/secondary.pub
}
```

以下是 Bootgen 命令：

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

此命令的输出为 secondary.pub.sha384。

阶段 1：分发 SPK 签名

可信人员向开发团队分发 SPK 签名。

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

此命令的输出为 secondary.pub.sha384.sig

阶段 2：在 FSBL 中使用 AES 进行加密

开发团队使用 Bootgen 创建所需数量的启动镜像。开发团队使用：

- 来自可信人员的 SPK 签名。
- SSK、SPK 和 SPKID

```
Stage2:  
{  
    [keysrce_cryption] bbram_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  
}
```

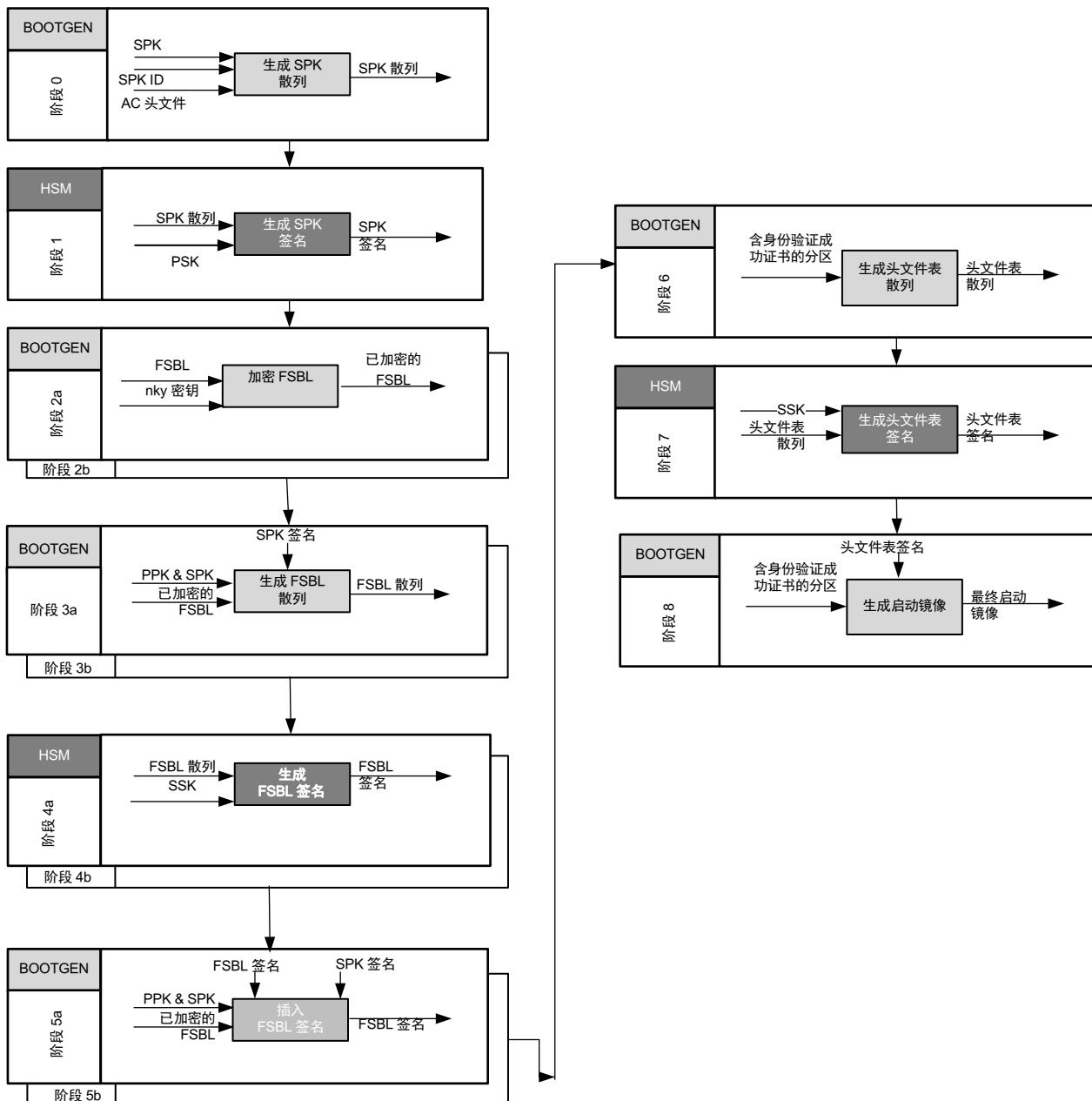
Bootgen 命令为：

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

使用 HSM 模式创建 Zynq 7000 SoC 器件启动镜像

下图提供了 AMD Zynq™ 7000 SoC 器件的 HSM 模式启动镜像图示。紧随在图示后提供了用于创建此启动镜像的步骤。

图 21：启动流程阶段 0 到 8



X21416-043022

使用 HSM 模式为 AMD Zynq™ 7000 SoC 器件创建启动镜像的过程与使用标准流程利用以下 BIF 文件创建启动镜像的过程类似。这些示例按需使用 OpenSSL 程序来生成散列文件。

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

阶段 0：为 SPK 生成散列

此阶段可生成 SPK 密钥的散列。

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

以下是 Bootgen 命令。

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

阶段 1：签署的 SPK 散列

此阶段会通过签署 SPK 散列来创建签名

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

或者通过使用以下 OpenSSL 程序来创建签名。

```
#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
```

阶段 2：创建分区二进制文件

此阶段会为该分区创建二进制文件。虽然本例是对分区进行加密，但即使此分区未加密，要创建二进制文件，也需要运行阶段 2。stage2.bif 如下所示。

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

Bootgen 命令如下所示。

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

输出为已加密的 fsbl_e.bin 文件。

阶段 3：生成分区散列

此阶段可生成不同分区的散列。

阶段 3a：生成 FSBL 散列

BIF 文件如下所示：

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

Bootgen 命令如下所示。

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

输出为散列文件 zynq_fsbl_0.elf.0.sha256。

阶段 3b：生成比特流散列

阶段 3b BIF 文件如下所示：

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

Bootgen 命令如下所示。

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

输出为散列文件 system.bit.0.sha256。

阶段 4：签署散列

此阶段会从创建的分区散列文件创建签名。

阶段 4a：签署 FSBL 分区散列

```
xil_rsa_sign.exe -gensig -sk secondary.pem -data zynq_fsbl_0.elf.0.sha256 -  
out zynq_fsbl_0.elf.0.sha256.sig
```

或者通过使用以下 OpenSSL 程序来创建签名。

```
#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
```

输出为签名文件 zynq_fsbl_0.elf.0.sha256.sig。

阶段 4b：签署比特流散列

```
xil_rsa_sign.exe -gensig -sk secondary.pem -data system.bit.0.sha256 -out
system.bit.0.sha256.sig
```

或者通过使用以下 OpenSSL 程序来创建签名。

```
#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
```

输出为签名文件 system.bit.0.sha256.sig。

阶段 5：插入分区签名

将以上创建的分区签名更改并插入身份验证证书。

阶段 5a：插入 FSBL 签名

stage5a.bif 如下所示。

```
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
}
```

Bootgen 命令如下所示。

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

经过身份验证的输出文件为 fsbl_e_ac.bin 和 efuseppkbts.txt。

阶段 5b：插入比特流签名

stage5b.bif 如下所示。

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

Bootgen 命令如下所示。

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

经过身份验证的输出文件为 system_e_ac.bin。

阶段 6：生成报头表散列

此阶段可生成报头表的散列。

stage6.bif 如下所示。

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

Bootgen 命令如下所示。

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

输出散列文件为 ImageHeaderTable.sha256。

阶段 7：生成报头表签名

此阶段可生成报头表签名。

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

或者通过使用以下 OpenSSL 程序来生成签名：

```
#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
```

输出为签名文件 ImageHeaderTable.sha256.sig。

阶段 8：将分区加以组合并插入报头表签名

stage8.bif 如下所示：

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

Bootgen 命令如下所示：

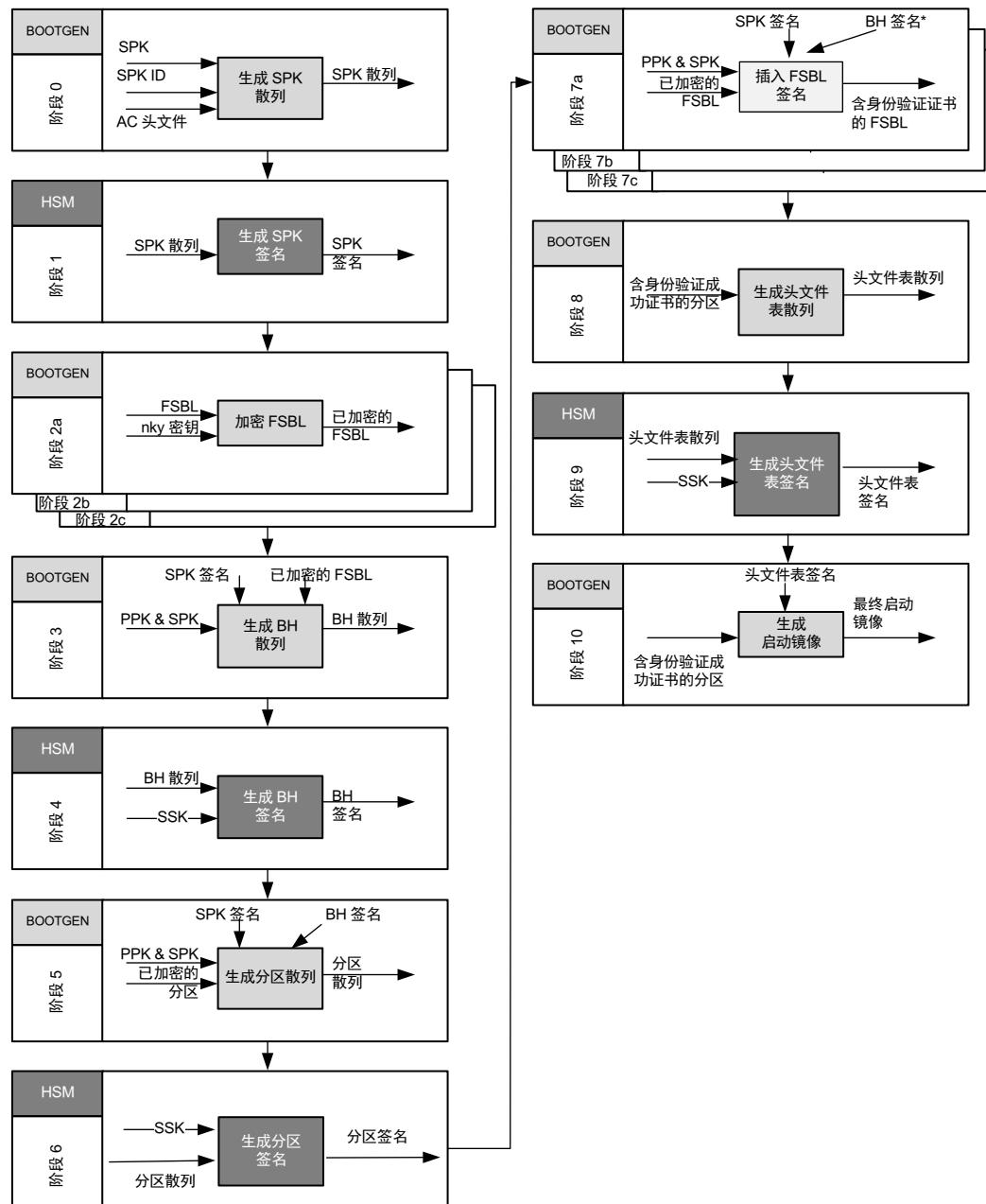
```
bootgen -image stage8.bif -w -o final.bin
```

输出为启动镜像文件 final.bin。

使用 HSM 模式创建 Zynq UltraScale+ MPSoC 器件启动镜像

下图提供了 HSM 模式启动镜像的图示。

图 22：启动流程的阶段 0 到阶段 10



X21547-043022

针对 AMD Zynq™ UltraScale+™ MPSoC 器件使用 HSM 模式创建启动镜像的过程与使用标准流程利用以下 BIF 文件创建启动镜像的过程类似。这些示例按需使用 OpenSSL 程序来生成散列文件。

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

```
[  
    bootloader,  
    destination_cpu=a53-0,  
    encryption=aes,  
    aeskeyfile=aes0.nky,  
    authentication=rsa  
] fsbl.elf  
  
[  
    destination_device=pl,  
    encryption=aes,  
    aeskeyfile=aes1.nky,  
    authentication=rsa  
] system.bit  
  
[  
    destination_cpu=a53-0,  
    authentication=rsa,  
    exception_level=e1-3,  
    trustzone=secure  
] bl31.elf  
  
[  
    destination_cpu=a53-0,  
    authentication=rsa,  
    exception_level=e1-2  
] u-boot.elf  
}
```

注释：要在 HSM 流程中使用 pmufw_image，请将 [pmufw_image] pmufw.elf 添加到以上 bif 文件中。在相似代码行上，必须在 stage2a bif（用于对 FSBL 进行加密）中添加此文件。流程其余部分相同。

阶段 0：为 SPK 生成散列

以下是来自 BIF 文件的代码片段。

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

以下是 Bootgen 命令：

```
bootgen -arch zynqmp -image stage0.bif -generate_hashes -w on -log error
```

阶段 1：签署 SPK 散列（对分区进行加密）

以下是使用 OpenSSL 生成 SPK 散列的代码片段：

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

此命令的输出为 secondary.pub.sha384.sig。

阶段 2a：对 FSBL 进行加密

在 BIF 文件中使用以下代码片段对 FSBL 进行加密。

```
Stage 2a:  
{  
    [keysrc_encryption] bbram_red_key  
    [  
        bootloader,destination_cpu=a53-0,  
        encryption=aes,  
        aeskeyfile=aes0.nky  
    ] fsbl.elf  
}
```

Bootgen 命令为：

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

阶段 2b：对比特流进行加密

生成以下 BIF 文件条目：

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

Bootgen 命令为：

```
bootgen -arch zynqmp -image stage2b.bif -o system_e.bin -w on -log error
```

阶段 3：生成启动报头散列

使用以下 BIF 文件生成启动报头散列：

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

Bootgen 命令为：

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

阶段 4：对启动报头散列进行签名

使用以下 OpenSSL 命令生成启动报头散列：

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

阶段 5：获取分区散列

在 BIF 文件中使用以下命令获取分区散列：

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

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

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

Bootgen 命令为：

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

针对每个比特流分区都会生成多个散列。欲知详情，请参阅 [使用外部存储器执行比特流身份验证](#)。

启动报头散列同样在此阶段 5 中生成；它不同于阶段 3 中生成的散列，因为在阶段 5 中不使用 `bh_auth_enable` 参数。如果需要，可在阶段 5 中添加该参数，但这并没有显著影响，因为使用阶段 3 生成的启动报头散列会在阶段 4 中签名，此签名仅在 HSM 模式流程中使用。

阶段 6：对分区散列进行签名

使用 OpenSSL 创建以下文件：

```
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
```

阶段 7：将分区签名插入身份验证证书

阶段 7a：在 BIF 文件中添加以下代码以插入 FSBL 签名：

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

Bootgen 命令如下所示：

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

阶段 7b：在 BIF 文件中添加以下代码以插入比特流签名：

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

Bootgen 命令为：

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

阶段 7c：在 BIF 文件中添加以下代码以插入 U-Boot 签名：

```
stage7c:
{
    [ppkfile] primary.pub
    [spkfile] secondary.pub
    [spksignature]secondary.pub.sha384.sig
    [bhsignature]boohader.sha384.sig
    [
        destination_cpu=a53-0,
```

```
        authentication=rsa,
        exception_level=el-2,
        presign=u-boot.elf.0.sha384.sig
    ] u-boot.elf
}
```

Bootgen 命令为：

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

阶段 7d：在 BIF 文件中输入以下代码以插入 ATF 签名：

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

Bootgen 命令为：

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

阶段 8：将分区加以组合并获取报头表散列

在 BIF 文件中输入以下代码：

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

Bootgen 命令为：

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

阶段 9：签署报头表散列

使用 OpenSSL 生成以下文件：

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

阶段 10：将分区加以组合并插入报头表签名

在 BIF 文件中输入以下代码：

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

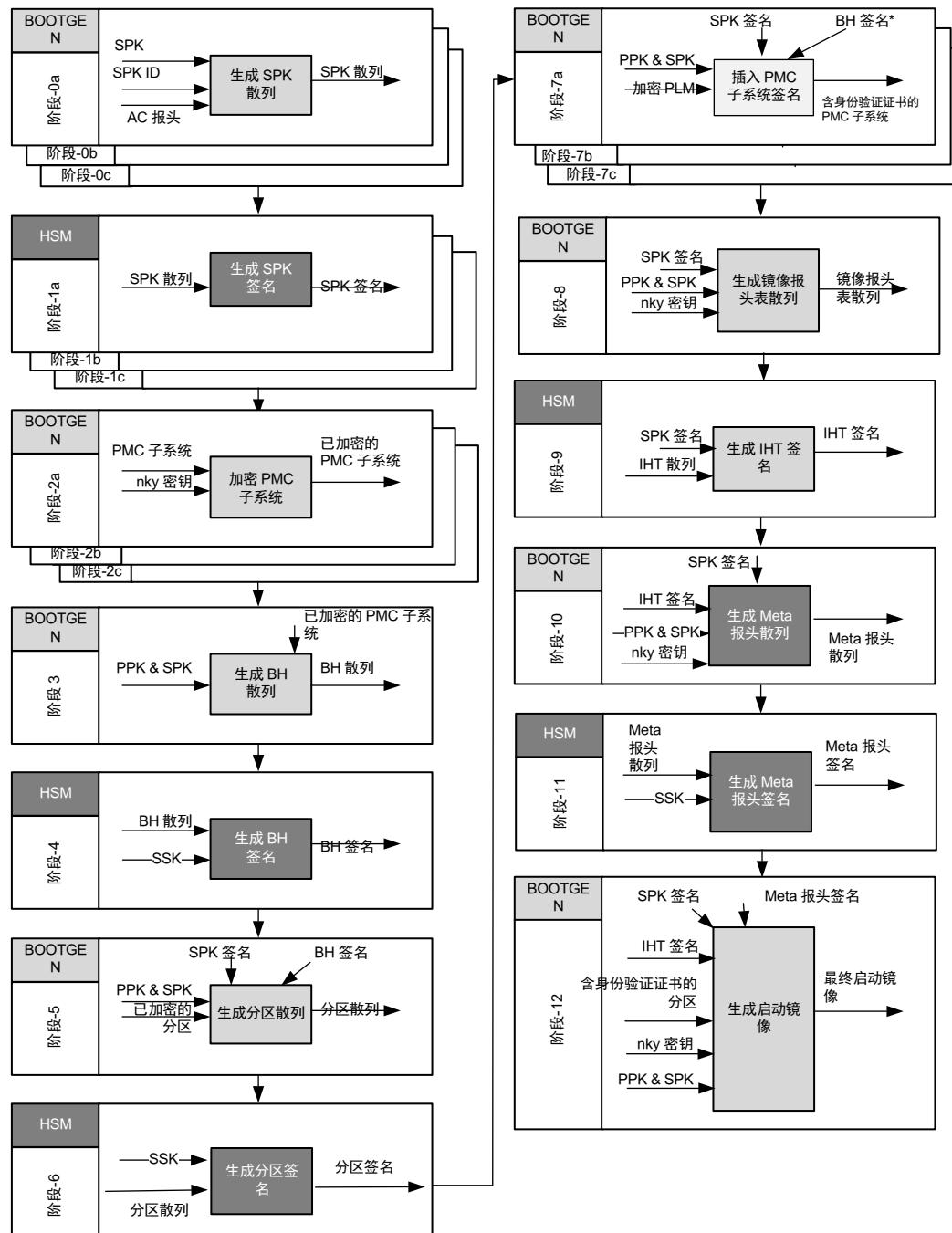
Bootgen 命令为：

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

使用 HSM 创建 Versal 器件启动镜像

下图提供了 Versal 器件的 HSM 模式启动镜像图示。

图 23：启动流程的阶段 0 到阶段 12



X21547-111020

注释：PMC 子系统包含 PLM 和 PMC_CDO。

生成 PDI

使用标准 BIF 生成 PDI。

```
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 = bbram_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 = 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 = 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 = bbram_red_key,
            aeskeyfile = gen_keys/key1.nky,
            dpacm_enable,
```

```
    file = images/gen_files/lpd_data.cdo
}
partition
{
    id = 0x0B, core = psm,
    authentication = rsa,
    pskfile = 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 = 0x420c0003
partition
{
    id = 0x08, type = cdo,
    authentication = rsa,
    pskfile = 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,
    pskfile = 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 模式步骤

阶段 0：生成 SPK 散列

生成 SPK1 的散列：

```
command : bootgen -arch versal -image stage0-SPK1.bif -generate_hashes -w  
on -log error  
  
stage0_SPK1:  
{  
    spkfile = rsa-keys/SPK1.pub  
}
```

生成 SPK2 的散列：

```
command : bootgen -arch versal -image stage0-SPK2.bif -generate_hashes -w  
on -log error  
  
stage0_SPK2:  
{  
    spkfile = rsa-keys/SPK2.pub  
}
```

生成 SPK3 的散列：

```
command : bootgen -arch versal -image stage0-SPK3.bif -generate_hashes -w  
on -log error  
  
stage0_SPK3:  
{  
    spkfile = rsa-keys/SPK3.pub  
}
```

阶段 1：对 SPK 散列进行签名

对已生成的散列进行签名：

```
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
```

阶段 2：创建分区二进制文件

此阶段会为该分区创建二进制文件。虽然本例是对分区进行加密，但即使此分区未加密，要创建二进制文件，也需要运行阶段 2。

加密分区 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,  
        keysrc = bbram_red_key,  
        aeskeyfile = encr_keys/pmcdata.nky  
        dpacm_enable  
        file = images/gen_files/pmc_data.cdo  
    }  
}  
}
```

加密分区 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  
        }  
    }  
}
```

加密分区 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
}
}
}
```

加密分区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 = bbram_red_key,
            aeskeyfile = encr_keys/key5.nky,
            dpacm_enable,
            file = images/gen_files/fpd_data.cdo
        }
    }
}
```

加密分区5

```
command : bootgen -arch versal -image stage2d.bif -o subsystem_e.bin -w on -log
error

stage2d:
{
    image
    {
        name = ss, id = 0x1c000033
        partition
        {
            id = 0x0D, type = cdo,
            encryption = aes,
            keysrc = bbram_red_key,
            aeskeyfile = encr_keys/key6.nky,
            dpacm_enable,
            file = images/gen_files/subsystem.cdo
        }
    }
}
```

阶段3：生成启动报头散列

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

stage3:
{
    boot_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
    }
}
```

阶段4：对启动报头散列进行签名

对已生成的散列进行签名：

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

阶段5：生成分区散列

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

stage5:
{
    bhsignature = boohandler.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
        }
    }
}
```

阶段 6：对分区散列进行签名

```
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
```

阶段7：将分区签名插入身份验证证书

插入分区1签名：

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

stage7a:  
{  
    bhsignature = botheader.sha384.sig  
    boot_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  
        }  
    }  
}
```

插入分区2签名：

```
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  
        }  
    }  
}
```

插入分区3签名：

```
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
    }
}
}
```

插入分区4签名：

```
command : bootgen -arch versal -image stage7c.bif -o fpd_e_ac.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
        }
    }
}
```

插入分区5签名：

```
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
        }
    }
}
```

阶段8：生成镜像报头表散列

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

stage8a:
{
    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}
}
```

阶段 9：对镜像报头表散列进行签名

对已生成的散列进行签名：

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

阶段 10：生成 Meta 报头散列

```
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}
}
```

阶段 11：对 Meta 报头散列进行签名

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

阶段 12：将分区进行组合，并插入报头签名

构建完整 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}
}
}
```

注释：如果使用 ecdsa 签名，以下提供了使用 openssl 的 ecdsa-p384 示例。

假设 secondary.pub.sha384 是 Bootgen 为给定 SPK 生成的哈希值，下面是使用 PSK primary.pem 生成 Bootgen 可用签名的脚本。

```
#!/bin/bash
ecdsa-p384-sign() {
cp $2 $2.hash
truncate -s 48 $2.hash
openssl pkeyutl -sign -inkey $1 -pkeyopt digest:sha3-384 -out $2.der -in
$2.hash
r=$(openssl asn1parse -in $2.der -inform DER | sed -n 2p | sed -n
's/.*\INTEGER.*:\(\.*\)/0000000000000000\1/p' | sed -n 's/.*\(\.\{96\}\)
\.\*/\1/p')
s=$(openssl asn1parse -in $2.der -inform DER | sed -n 3p | sed -n
's/.*\INTEGER.*:\(\.*\)/0000000000000000\1/p' | sed -n 's/.*\(\.\{96\}\)
\.\*/\1/p')
padding=$(head -c 832 /dev/zero | LC_ALL=C tr "\000" "00")
echo -n $r$s$padding > $3
}
ecdsa-p384-sign primary.pem secondary.pub.sha384 secondary.pub.sha384.sig
```

SSIT 支持

堆叠硅片互联技术（SSI 技术）用于打破摩尔定律的限制，提供所需功能以满足最严苛的设计要求。

采用 SSI 技术的器件由多个超级逻辑区域 (SLR) 组成，其中每个 SLR 均为一颗裸片。SLR0 也称为 Master SLR（主 SLR），它是底部裸片。SLR1 是自下而上第二颗裸片，SLR2 是自下而上第三颗，以此类推。AMD Versal™ 自适应 SoC SSI 技术变体可包含最多 4 个 SLR。

每个 SLR 都有其自己的平台管理控制器 (PMC) 和可编程逻辑 (PL) 区域，就像单片器件 Versal 自适应 SoC 一样，但 slave SLR（从 SLR）没有 AI 引擎和处理器系统 (PS) 区域。

要在此器件上启动的最终 PDI 是所有 PDI 的 PDI。由于每个 SLR 都有其自己的 PMC 块，因此每个 SLR 都会随一个 PDI 一起启动，此 PDI 集成在主 PDI 内。

注释：每个 SLR 中的 PLM elf 都应相同。

含 SSI 技术器件的 Versal 自适应 SoC 的 BIF 不同于其单片器件变体。以下是含 2 个 SLR 器件的 bif 示例。

注释：以下整个 BIF 代码块都纳入单个文件。Bootgen 会读取多个 BIF 块，并基于 BIF 标签创建各自的 PDI。在主 BIF 中引用这些 BIF 标签，Bootgen 会基于主 BIF 将各 PDI 合并为主 PDI。此主 PDI 即足以启动 SSI 技术器件。

注释：从 SLR 搭配指示下游连接的特殊 smap_width=0 选项一起使用，且不得更改。

适用于单从 SLR 器件的 SSI 技术非安全 BIF 示例：

HBM 器件示例

```
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  
    }  
}  
}
```

适用于双从 SLR 器件的 SSI 技术非安全 BIF 示例：

```
// 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 = pmcdtata, 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 }
}
```

注释：SSI 技术器件支持 MCS 格式启动镜像/PDI 生成。为从接口生成的中间 PDI 始终采用二进制格式。生成的最终 PDI 将采用 mcs 格式（前提是已选中此格式）。

适用于三从 SLR 器件的 SSI 技术身份验证 BIF 示例：

```
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}
    pskfile = 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 支持

如 [第 5 章：启动时间安全](#) 中所述，仅限 FPGA 的器件现场部署时还需开展安全性维护工作。AMD 工具可提供嵌入式 IP 模块来实现编程逻辑中包含的加密和身份验证。Bootgen 为从 7 系列起的 FPGA 家族器件扩展了安全镜像创建（加密和/或身份验证）支持。本章提供了有关如何使用 Bootgen 来对比特流进行加密和身份验证的部分示例，并对这些示例进行了详细说明。在 Bootgen 独立安装中可为 FPGA 提供 Bootgen 支持。

注释：仅支持来自 7 系列器件和更高版本的器件的比特流。

加密和身份验证

AMD 7 系列 FPGA 使用基于 PL 的嵌入式散列消息认证码 (HMAC) 和含密码分组链接 (CBC) 模式的高级加密标准 (AES) 模块。对于 UltraScale 器件和更高版本的器件，则使用 AES-256/Galois 计数器模式 (GCM)，无需 HMAC。

加密示例

为创建加密比特流，在 BIF 中使用 `aeskeyfile` 属性指定了 AES 密钥文件。必须根据需加密的 BIF 文件中列出的比特流来指定 `encryption=aes` 属性。

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

BIF 文件如下所示：

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

身份验证示例

用于对 FPGA 比特流进行身份验证的 Bootgen 命令如下所示：

```
bootgen -arch fpga -image all.bif -o rsa.bit -w on -log error
```

BIF 文件如下所示：

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

族密钥或模糊密钥

为支持模糊密钥加密，必须向 AMD 支持人员注册，并请求获取目标器件家族的族密钥文件。尝试执行模糊密钥加密前，必须将此文件的存储路径作为 `bif` 选项来传递。请联系 `secure.solutions@xilinx.com` 以获取族密钥。

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

`aeskey` 文件样本如下图所示。

图 24: AES 密钥样本

```
Device xckull5;  
EncryptKeySelect BBRAM;  
KeyObfuscate 94da9014cb2203f502f81d14fa2471f4a8902b16d9d408c9c66db214c1640db7, 0;  
StartIvObfuscate c485144e397a92081ad20c867a005272, 0;  
Key0 dcd2e72ad1b281ecca5e0790b65b94090ec1c8fc010eb01e56717345df4c7010, 0;  
StartIv0 3fe826e5495db1bdaf0c2ca2e8640911, 0;  
KeyObfuscate 967a6d1ecccefdd1990241007de18f41d69ca7231852c0061fb6c78e204c5f3, 1;  
StartIvObfuscate 7ab9a7ca88474d7f95ed1b548523451b, 1;  
Key0 af84947a9cc256c090d5aelc53ed3fd33bb553d7039e445829ba4cffbe56ffe3, 1;  
StartIv0 a50026e212363eld7lfa6f4fb540ce42, 1;
```

HSM 模式

对于量产 (Production) 版本，FPGA 使用 HSM 模式，并且还可使用标准 (Standard) 模式。

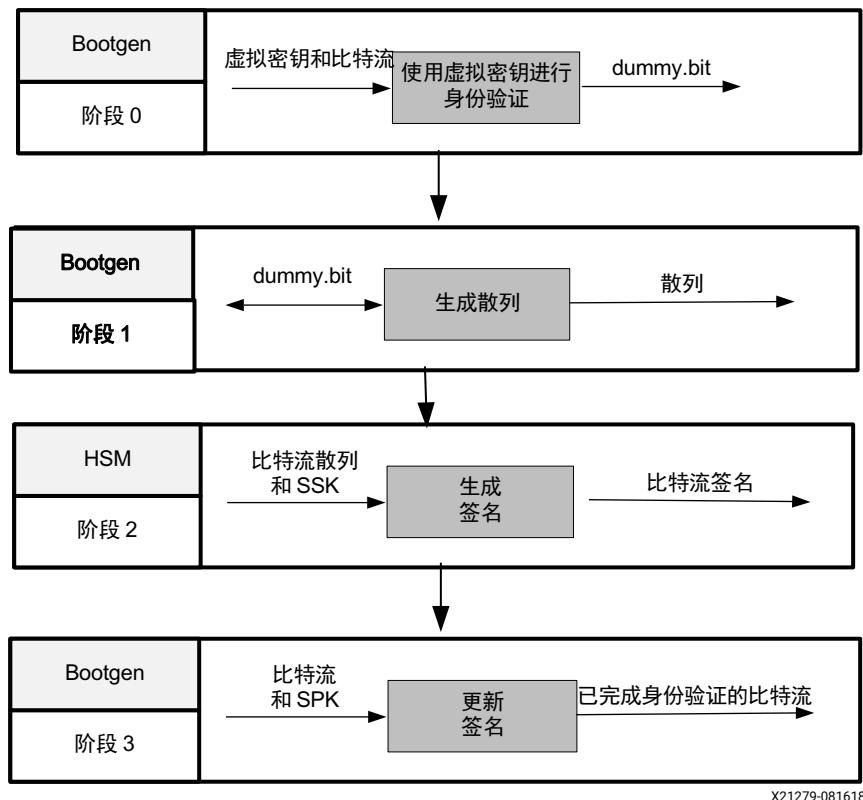
标准模式

标准模式可生成包含嵌入式身份验证签名的比特流。在此模式下，秘密密钥应可供用户用于生成经过身份验证的比特流。按如下方式运行 Bootgen：

```
bootgen -arch fpga -image all.bif -o rsa_ref.bit -w on -log error
```

下列步骤描述了如何在 HSM 模式下生成经过身份验证的比特流，其中秘密密钥由安全团队进行维护，不可供用户使用。下图显示了 HSM 模式流程：

图 25：HSM 模式流程



X21279-081618

阶段 0：使用虚拟密钥进行身份验证

这是针对给定比特流的一次性任务。对于阶段 0，Bootgen 可生成 `stage0.bif` 文件。

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

`stage0.bif` 的内容如下所示。如需了解格式，请参阅后续阶段。

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

注释：经过身份验证的比特流包含报头、实际比特流、签名和脚注。创建此 `dummy.bit` 的目的是以虚拟签名获取与经过验证的比特流格式相同的比特流。现在，将虚拟比特文件提供给 Bootgen 时，它会计算签名并在偏移处插入，以提供经过身份验证的比特流。

阶段 1：生成散列

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

Stage1.bif 如下所示：

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

阶段 2：对散列 HSM 进行签名

这里使用 OpenSSL 进行演示。

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

阶段 3：使用真实签名来更新 RSA 证书

Stage3.bif 如下所示：

```
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  
}
```

注释：公钥摘要必须烧录到 eFUSE 中，在 HSM 模式下的阶段 3 的 `rsaPubKeyInfo.pem.nky` 文件中可找到此公钥摘要。

同时使用身份验证和加密的 HSM 流程

阶段 0：使用虚拟密钥对纯比特流进行加密和身份验证。如果需要密钥滚动，请添加 `keylife` 参数。

您可以提供 `.nky` 文件，或者 Bootgen 可以生成包含加密密钥的 `.nky` 文件。Bootgen 不支持生成 AES 模糊密钥。密钥滚动功能需要 `keylife` 参数。

```
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
```

完成此步骤后，如果启用了加密，则会生成 `.nky` 文件。此文件包含所有密钥。

阶段 1：生成散列

请参阅以下代码示例。

```
the_ROM_image:  
{  
[authentication=rsa] auth-encrypt-system.bit  
}  
  
bootgen -arch fpga -image stage1.bif -generate_hashes -log info
```

阶段 2：对散列 HSM 进行签名

这里使用 OpenSSL 进行演示。

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

您可以使用 HSM 服务器对散列进行签名。对于 SSI 技术器件，为每个超级逻辑区域 (SLR) 生成签名。以下示例显示了用于为具有四个 SLR 的器件生成签名的代码。

```
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
```

阶段 3：使用真实签名来更新 RSA 证书

请参阅以下代码示例。

```
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
```

注释：对于 SSI 技术器件，使用 presign=<first presign filename>:<number of total presigns>。例如，具有四个 SLR 的器件应使用 <first presign filename:4>。

用例与示例

以下是 Bootgen 的典型用例和示例。部分用例较为复杂，需要明确的指示信息。如需了解这些典型用例和示例的详细定义，请参阅 [属性](#)。

Zynq MPSoC 用例

不同核上的简单应用启动

以下示例显示了如何使用在不同的核上运行的应用来创建启动镜像。`pmu-fw.elf` 由 BootROM 加载。`fsbl-a53.elf` 为启动加载程序，加载到 A53-0 核上。`app-a53.elf` 由 A53-1 核执行，`app-r5.elf` 则由 r5-0 核执行。

```
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  
}
```

由 BootROM 加载的 PMU 固件

此示例显示如何使用由 BootROM 加载的 `pmu-fw.elf` 来创建启动镜像。

```
the_ROM_image:  
{  
    [pmufw_image] pmu-fw.elf  
    [bootloader, destination_cpu=a53-0] fsbl-a53.elf  
    [destination_cpu=r5-0] app-r5.elf  
}
```

此示例显示如何使用由 BootROM 加载的 `pmu-fw.elf` 来创建启动镜像。如果 PMU 固件是使用 `[pmufw_image]` 属性指定的，则不会将 PMU 固件作为独立分区来处理。而是将其追加到 FSBL 之后。FSBL 与 PMU 固件一起组合为单一大型分区。因此，您在 Bootgen 的 log 日志中也看不到此 PMU 固件。

由 FSBL 加载的 PMU 固件

此示例显示如何使用由 FSBL 加载的 pmu_fw.elf 来创建启动镜像。

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

注释: Bootgen 还可将提供给 [bootloader] 的选项应用于 [pmufw_image]。[pmufw_image] 不使用任何其他参数。

启动 Linux

此示例演示了如何在 AMD Zynq™ UltraScale+™ MPSoC (arch=zynqmp) 上启动 Linux。

- fsbl_a53.elf 为启动加载程序，在 a53-0 上运行。
- pmu_fw.elf 由 FSBL 加载。
- bl31.elf 为 Arm® 可信固件 (ATF)，在 el-3 上运行。
- U-Boot 程序 uboot 在 a53-0 的 el-2 上运行。
- Linux 镜像 image.ub 置于偏移 0x1E40000 处，并在 0x10000000 处加载。

```
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  
}
```

加密流程：BBRAM 红密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用 BBRAM 中存储的红密钥来创建应用：

```
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  
}
```

加密流程：eFUSE 中存储的红密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用 eFUSE 中存储的红密钥来创建应用。

```
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  
}
```

加密流程：eFUSE 中存储的黑密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用 eFUSE 中存储的 efuse_blk_key 来创建应用。针对 FSBL 还会启用身份验证。

```
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  
}
```

注释：使用黑密钥加密时，启动镜像身份验证是必需的。

加密流程：启动报头中存储的黑密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用启动报头中存储的 bh_blk_key 来创建应用。针对 FSBL 还会启用身份验证。

```
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
[keysr 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
}
```

注释：使用黑密钥加密时，需启动镜像身份验证。

加密流程：eFUSE 中存储的灰密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用 eFUSE 中存储的 efuse_gry_key 来创建应用。

```
the_ROM_image:
{
    [keysr 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
}
```

加密流程：启动报头中存储的灰密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用启动报头中存储的 bh_gry_key 来创建应用。

```
the_ROM_image:
{
    [keysr 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
}
```

运行密钥

此示例演示了如何创建启动镜像并在其中为 FSBL 启用加密，以及如何使用 eFUSE 中存储的红密钥来创建应用。

```
the_ROM_image:
{
    [fsbl_config] opt_key
    [keysrccryption] 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
}
```

使用运行密钥来保护开发环境中的器件密钥

以下步骤提供了如下场景下的解决方案：存在 2 支开发团队，团队 A（安全团队）管理红色秘密密钥，团队 B（非安全团队）协作构建加密镜像，但不共享红色秘密密钥。团队 B 构建加密镜像用于开发和测试。但，后者无权访问红色秘密密钥。

团队 A 使用器件密钥（通过使用 `Op_key` 选项）来对启动加载程序进行加密，并将加密后的启动加载程序交付给团队 B。团队 B 使用 `Op_key` 对所有其他分区进行加密。

团队 B 使用其创建的已加密分区以及团队 A 为其提供的已加密的启动加载程序，并使用 Bootgen 将所有一切都缝合在一起，组成单个 boot.bin。

以下过程描述了用于构建镜像的步骤：

步骤 1

在初始步骤中，团队 A 使用器件密钥和 `opt_key` 选项对启动加载程序进行加密，并将加密后的启动加载程序交付给团队 B。现在，团队 B 即可一次性创建完整的镜像，其中包含所有分区和已加密的启动加载程序（使用“运行密钥”作为“器件密钥”）。

1. 使用器件密钥来对启动加载程序进行加密：

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

stage1.bif 示例:

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

对应 stage1 的 aes.nky 示例:

```
Device xc7z020c1g484;  
Key 0 AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;  
IV 0 F7F8FDE08674A28DC6ED8E37;  
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
```

2. 将加密后的启动加载程序和其余分区（使用“运行密钥”作为“器件密钥”）连接在一起构成完整镜像：

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

stage2.bif 示例:

```
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  
}
```

对应 stage2 的 aes-opt.nky 示例:

```
Device xc7z020c1g484;  
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;  
IV 0 F7F8FDE08674A28DC6ED8E37;
```

步骤 2

在初始步骤中，团队 A 使用器件密钥和 opt_key 选项对启动加载程序进行加密，然后将加密后的启动加载程序交付给团队 B。现在，团队 B 即可为每个分区单独创建经过加密的镜像（使用“运行密钥”作为“器件密钥”）。最后，团队 B 可使用 Bootgen 来将所有经过加密的分区和经过加密的启动加载程序连接起来，获得完整的镜像。

1. 使用器件密钥来对启动加载程序进行加密：

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

stage1.bif 示例：

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

对应 stage1 的 aes.nky 示例：

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

2. 使用“运行密钥”作为“器件密钥”，对其余分区进行加密：

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

stage2a.bif 示例：

```
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
```

stage2b.bif 示例：

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

对应 stage2a 和 stage2b 的 aes-opt.nky 示例：

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

3. 使用 Bootgen 将以上示例连接在一起，构成完整的镜像：

```
bootgen -arch zynqmp -image stage3.bif -o boot.bin -w on -log error
```

stage3.bif 示例：

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

注释：aes.nky 的 opt_key 与 aes-opt.nky 中的 Key 0 相同，两个 nky 文件中的 IV 0 也必须相同。

单分区镜像

该功能可为 Bootgen 根据 U-Boot 提示所创建的单分区（非比特流）镜像的身份验证和/或解密提供支持。

注释：该功能不支持含多个分区的镜像。

用于加载安全镜像的 U-Boot 命令

```
zynqmp secure <srcaddr> <len> [key_addr]
```

此命令会验证位于地址 \$src 的安全镜像的长度是否是 \$len bytes。如需使用用户密钥来进行解密，可指定可选 key_addr。

仅执行身份验证的用例

要在 U-Boot 处仅使用身份验证，请使用 bif 创建已经过身份验证的镜像，如以下示例所示。

1. 创建在 U-Boot 处执行身份验证的单分区镜像。

注释：如果提供 elf 文件，它不应包含多个可加载节。如果 elf 文件包含多个可加载节，则应将输入转换为 .bin 格式，并在 bif 中提供 .bin 作为输入。以下提供了 bif 示例：

```
the_ROM_image:  
{  
    [pskfile]rsa4096_private1.pem  
    [sskfile]rsa4096_private2.pem  
    [auth_params] ppk_select=1;spk_id=0x1  
    [authentication = rsa]Data.bin  
}
```

2. 生成镜像后，请将经过身份验证的镜像下载到 DDR。
3. 执行 U-Boot 命令以对安全镜像进行身份验证，如以下示例所示。

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

4. U-Boot 会在成功完成身份验证后返回实际分区的起始地址。发生操作失败时，U-Boot 会打印错误代码。如果 RSA_EN eFUSE 已烧录，那么必须执行镜像身份验证。启用 eFUSE RSA 时不支持启动报头身份验证。

仅执行加密的用例

如果镜像仅加密，则不支持器件密钥。未启用身份验证时，仅支持 KUP 密钥解密。

身份验证流程

此示例显示了如何创建启动镜像并在其中为 FSBL 启用身份验证，以及如何创建应用并在其中启用启动报头身份验证以绕过 PPK 散列验证：

```
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  
}
```

含 SHA-3 eFUSE RSA 身份验证和 PPK0 的 BIF 文件

此示例演示了如何创建启动镜像并在其中为 FSBL 启用身份验证，以及如何使用 eFUSE 身份验证来创建应用。这是默认选择。在此进程中，启动镜像中的 PPK 散列使用来自 eFUSE 的散列进行验证。

```
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

此示例显示了如何创建针对 zynqmp (AMD Zynq™ UltraScale+™ MPSoC) 就地执行的启动镜像：

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

请参阅 [xip_mode](#) 以获取有关此命令的更多信息。

使用“Offset”属性拆分

此示例有助于理解 offset 属性的拆分方式。

```
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=0x6000000]\ hello.elf  
}
```

对分区指定偏移 (offset) 时，启动镜像中该分区的地址即从给定偏移开始。为弥补当前分区注明的偏差与先前分区之间的间隙，Bootgen 会向先前分区追加 0xFF。因此，现在对同一个分区尝试执行拆分时，启动镜像应根据该分区的地址进行拆分，即此例中所提及的偏移。因此，您可在拆分后的分区输出中看到填充后的 0xFF。

Versal 自适应 SoC 用例

对于 AMD Versal™ 自适应 SoC，AMD Vivado™ 可生成启动镜像，称为可编程器件镜像 (PDI)。此 AMD Vivado™ 生成的 PDI 包含启动加载程序软件可执行文件，即 PLM，以及 PL 相关组件和支持性数据文件。基于工程和 CIPS 配置，Vivado 会创建 BIF 文件并调用 Bootgen 来创建 PDI。此 BIF 可作为 XSA 的一部分导出至 AMD Vitis™ 之类的软件工具。随后，可使用所需属性为所需分区修改此 BIF。确保在 BIF 文件中，`id_code` 和 `extended_id_code` 相关的代码行均保留不变。此信息是 Bootgen 生成的 PDI 镜像所必需的。

如果要手动编写 BIF，请参阅由 Vivado 为相同器件生成的 BIF，确保将 `id_code` 和 `extended_id_code` 相关的代码行都添加到手动编写的 BIF 中。由 Vivado 生成的 BIF 样本如下所示：

```
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  
}  
}  
}
```

注释：Vivado 工程中生成的 BIF 文件位于 <vivado_project>/<vivado_project>.runs/impl_1/<Vivado_project>_wrapper.pdi.bif 中。

启动加载程序和 PMC_CDO

此示例演示了如何将启动加载程序与 PMC_CDO 搭配使用。

```
all:  
{  
    id_code = 0x04ca8093  
    extended_id_code = 0x01  
  
    init = reginit.ini  
    image  
{  
        {type=bootloader, file=PLM.elf}  
        {type=pmcdata, file=pmc_cdo.bin}  
    }  
}
```

启动加载程序、PMC_CDO 和加载地址

此示例显示了如何将启动加载程序与 PMC_CDO 和加载地址搭配使用。

```
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}
    }
}
```

为启动加载程序启用校验和

本示例演示了如何在使用启动加载程序时启用校验和。

```
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}
    }
}
```

启动加载程序、PMC_CDO、PL CDO 和 NPI

此示例演示了如何将启动加载程序与 PMC_CDO 和 NPI 搭配使用。

```
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 }  
}  
}
```

启动加载程序、PMC_CDO、PL CDO、NPI、PS CDO 和 PS ELF

此示例演示了如何将启动加载程序与 PMC_CDO、NPI、PS CDO 和 PS ELF 搭配使用。

```
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 引擎配置和 AI 引擎分区

本示例演示了如何配置 AI 引擎启动镜像和 AI 引擎分区。

```
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
        }
    }
}
```

注释: 通过合并不同 CDO 以在 PDI 中形成单一分区。

向现有 PDI 追加新分区

本示例演示了如何向现有 PDI 追加新分区。

1. 使用 Vivado 生成的 PDI (base.pdi)。
2. 追加 dtb、uboot 和 bl31 应用以创建新 PDI。

```
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 身份验证示例

此示例演示了如何使用 RSA 身份验证。

```
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, pskfile = ./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, pskfile = ./PSK1.pem, sskfile = ./SSK5.pem, revoke_id = 0x5,
         file = ./fpd_data.cdo}
    }
}
```

ECDSA 身份验证示例

此示例演示了如何使用 ECDSA 身份验证。

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

AES 加密示例

此示例演示了如何使用 AES 加密。

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

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

含密钥滚动的 AES 加密示例

此示例演示了如何使用含密钥滚动的 AES 加密。

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

    image
    {
        {
            type=bootloader,
            encryption=aes,
            keysrc=bbram_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
        }
    }
}
```

含多个密钥源的 AES 加密示例

此示例演示了如何针对不同分区使用不同密钥源。

```
all:
{
    bh_keyfile = ./PUF4K_KEY.txt
    puf_file = ./PUFHD_4K.txt
    bh_kek_iv = ./blk_iv.txt
    bbram_kek_iv = ./bbram_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=bbram_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=bbram_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 加密和身份验证示例

此示例演示了如何使用 AES 加密和身份验证。

```
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, pskfile = ./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, pskfile = ./PSK1.pem, sskfile = ./SSK5.pem,
     file = ./fpd_data.cdo
    }
}
```

替换现有 PDI 中的 PLM

此示例演示了替换现有 PDI 中的 PLM 的步骤。

1. 使用 Vivado 生成的 PDI (base.pdi)。
2. 通过替换 base PDI 中的 PLM (启动加载程序) 来创建新的 PDI。

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

Bootgen 会将启动加载程序 plm.elf 替换为新的 plm_v1.elf。

用于创建 PDI 的 Bootgen 命令示例

使用以下命令来创建 PDI。

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

替换 SSI 技术 PDI 中的 PLM 和 PMC CDO

Bootgen 支持取 PDI 作为输入来替换单片器件中的 PLM、PMC CDO 和 PSM。

考虑到 SSI 技术器件及未来需求，现在取 bif 作为输入替代启动镜像来完成替换功能。

此示例演示了替换 PLM 和 PMC DATA 的步骤。

1. 提取 Vivado 生成的 bif (base.bif)。

2. 通过替换 base bif 中的 PLM (启动加载程序) 来创建新的 PDI。

```
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 会将相应的 slr 启动加载程序 plm.elf 替换为新的 plm_v1.elf 并将 pmcdata 替换为 pmc_data-v1.cdos。

用于创建 PDI 的 Bootgen 命令示例

使用以下命令来创建 PDI。

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

演示 emmc/mmc 作为含多个文件系统分区的辅助启动器件的 BIF 示例

指向 emmc/mmc 中的首个文件系统分区的主启动 BIF

```
new_bif:
{
    boot_device { mmc,address=0x1000A }
    id_code = 0x14ca8093
    extended_id_code = 0x01
    id = 0x1
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0x2
            type = bootloader
            file = ./plm.elf
        }
        partition
        {
            id = 0x3
            type = pmcdata,load=0xF2000000
            file = ./pmc_data.cdo
        }
    }
}
```

```
image
{
    name = lpd_subsys
    id = 0x4210002
    partition
    {
        id = 0x4
        type = cdo
        file = ./lpd_data.cdo
    }
    partition
    {
        id = 0x5
        core = psm
        file = ./psm-fw.elf
    }
}
image
{
    name = fpd_subsys
    id = 0x420c003
    partition
    {
        id = 0x6
        type = cdo
        file = ./fpd_data.cdo
    }
}
```

指向 emmc/mmc 中的其他文件系统分区的主启动 BIF

```
new_bif:
{
    boot_device { mmc,address=0x2000A }
    id_code = 0x14ca8093
    extended_id_code = 0x01
    id = 0x1
    image
    {
        name = pmc_subsys
        id = 0x1c000001
        partition
        {
            id = 0x2
            type = bootloader
            file = ./plm.elf
        }
        partition
        {
            id = 0x3
            type = pmcdata,load=0xF2000000
            file = ./pmc_data.cdo
        }
    }
    image
    {
        name = lpd_subsys
        id = 0x4210002
        partition
        {
            id = 0x4
```

```
    type = cdo
    file = ./lpd_data.cdo
}
partition
{
    id = 0x5
    core = psm
    file = ./psm_fw.elf
}
}
image
{
    name = fpd_subsys
    id = 0x420c003
    partition
    {
        id = 0x6
        type = cdo
        file = ./fpd_data.cdo
    }
}
```

BIF 属性参考

注释：如果使用以下任意 BIF 属性作为文件名，Bootgen 将无法识别该文件名。请使用绝对路径或相对路径绕过该属性。

示例：file=./Image

由于“image”是 bif 属性，因此需要提供路径以帮助 Bootgen 加以区分。

aarch32_mode

语法

- 对于 AMD Zynq™ UltraScale+™ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

```
{aarch32_mode, file=<partition>}
```

描述

指定二进制文件将以 32 位模式执行。

注释：Bootgen 会从 .elf 文件自动检测处理器的执行模式。此项仅对二进制文件有效。

实参

指定的分区。

示例

- 对于 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  
}
```

- 对于 Versal 自适应 SoC:

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

注释: *base.pdi 即为 Vivado 所生成的 PDI。

aeskeyfile

语法

- 对于 Zynq 器件和 FPGA:

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

- 对于 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

```
{ aeskeyfile = <keyfile name>, file = <filename> }
```

描述

指向 AES 密钥文件的路径。密钥文件包含用于加密分区的 AES 密钥。此密钥文件的内容必须写入 eFUSE 或 BBRAM。如果指定路径内不存在密钥文件，则将由 Bootgen 生成新密钥用于加密。

注释: 仅适用于 Zynq UltraScale+ MPSoC: 在 BIF 文件内需指定多个密钥文件。在使用的所有 NKY 文件中，Key0、IV0 和 Key Opt 都应相同。如果任一 ELF 文件生成多个分区，则可使用来自唯一密钥文件的密钥对每个分区进行加密。请参阅以下示例。

实参

指定的文件名。

返回值

无

Zynq 7000 SoC 示例

fsbl.elf 和 hello.elf 分区均使用 test.nky 中的密钥进行加密。

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

密钥 (.nky) 文件样本 - test.nky

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

Zynq UltraScale+ MPSoC 示例

示例 1:

fsbl.elf 分区使用 test.nky 中的密钥加密, hello.elf 分区使用 test1.nky 中的密钥加密, app.elf 分区使用 test2.nky 中的密钥加密。样本 BIF - test_multiple.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  
}
```

示例 2:

假设 Bootgen 为 hello.elf 创建 3 个分区, 分别称为 hello.elf.0、hello.elf.1 和 hello.elf.2。样本 BIF - test_multiple.bif

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

其他信息:

- fsbl.elf 分区使用 test.nky 中的密钥加密。所有 hello.elf 分区都使用 test1.nky 中的密钥进行加密。
- 将名为 test1.1.nky 和 test1.2.nky 的密钥文件包含在与 test1.nky 相同路径内, 即可为每个 hello 分区指定专用密钥文件。
- hello.elf.0 使用 test1.nky
- hello.elf.1 使用 test1.1.nky
- hello.elf.2 使用 test1.2.nky
- 如果任一密钥文件 (test1.1.nky 或 test1.2.nky) 不存在, 那么 Bootgen 会生成该密钥文件。

- aeskeyfile 格式:

.nky 文件接受以下字段。

- Device: 使用 nky 文件的器件的名称。针对 Zynq 器件和 Zynq UltraScale+ MPSoC 均有效。
- Keyx 和 IVx: 此处 “x” 表示对应于 Key/IV 编号的整数，例如，Key0、Key1、Key2 ...、IV0、IV1、IV2...AES 密钥长度必须为 256 位，而 IV 密钥长度必须为 12 字节。Keyx 针对 Zynq 器件和 Zynq UltraScale+ MPSoC 均有效，但 IVx 仅针对 Zynq UltraScale+ MPSoC 有效。
- Key Opt: 可选密钥，可供您用于对启动加载程序的第一个块进行加密。仅针对 Zynq UltraScale+ MPSoC 有效。
- StartCBC - CBC Key: CBC 密钥长度必须为 128 位。仅针对 Zynq 器件有效。
- HMAC - HMAC Key: HMAC 密钥长度必须为 128 位。仅针对 Zynq 器件有效。
- Seed: 分区加密所需的初始种子，用于生成 Key/IV 对。AES 种子长度必须为 256 位。仅针对 Zynq UltraScale+ MPSoC 有效。
- FixedInputData: 与种子 (Seed) 一起用作为 “Counter Mode KDF”（计数器模式密钥衍生函数）的输入的数据。AES 固定输入数据长度必须为 60 字节。仅针对 Zynq UltraScale+ MPSoC 有效。

注释:

- Seed 必须随 FixedInputData 一起指定。
- 如果存在多个 key/iv 对，则无需种子。

Versal 自适应 SoC 示例

```
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
        }
        {
            type=cdo, encryption = aes,
            keysrc = efuse_red_key, aeskeyfile = key3.nky,
            file=fpd_data.cdo
        }
    }
}
```

alignment

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

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

设置字节对齐。对此分区进行填充以对齐到该值的倍数。该属性不能配合偏移使用。

实参

要对齐的字节数。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

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

注释: *base.pdi 即为 Vivado 所生成的 PDI。

auth_params

语法

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

描述

身份验证参数可指定其他配置，例如用于对启动镜像中的分区进行身份验证的 PPK 和 SPK。此 bif 参数的实参包括：

- ppk_select：选择要使用的 PPK。选项为 0（默认值）或 1。
- spk_id：指定可使用或撤销的 SPK。请参阅 [含增强型 RSA 密钥撤销的用户 eFUSE 支持](#)。默认值为 0x00。

注释：虽然报头与 FSBL 采用不同 SPK，但这两者将共享相同的 SPK ID。

如果仅使用 auth_params 字段，并且其中提供了 SPK ID，那么此 SPK ID 会传输至启动和应用分区。如果在启动和应用分区中都使用 SPK ID，那么启动/镜像报头分区中的 SPK ID 会被覆盖，并使用应用 SPK。这意味着 Bootgen 在确保报头与 FSBL 具有相同 SPK ID 的过程中，会选择为其提供的最新版本的 SPK ID。

- spk_select：用于区分 spk 和 user eFUSE。选项包括 spk-efuse（默认值）和 user_efuse。
- header_auth：在不对分区进行身份验证的情况下用于对报头进行验证。

注释：

1. ppk_select 针对每个镜像都唯一。
2. 每个分区都可有专用的 spk_select 和 spk_id。
3. spk-efuse id 在整个镜像内唯一，但 user-efuse id 可因分区而异。
4. 分区范围外的 spk_select/spk_id 用于报头以及不采用这些规格作为分区属性的任何其他分区。

示例

样本 BIF 1 - test.bif

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

样本 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  
}
```

样本 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  
}
```

样本 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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

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

描述

指定要进行身份验证的分区。

实参

- none: 分区不执行身份验证。这是默认值。
- rsa: 分区使用 RSA 算法执行身份验证。
- ecdsa-p384: 分区使用 ECDSA p384 曲线执行身份验证
- ecdsa-p521: 分区使用 ECDSA p521 曲线执行身份验证

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 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,  
            pskfile =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,  
            pskfile = PSK3.pem,  
            sskfile = SSK3.pem,  
            file = lpd_data.cdo  
        }  
        partition  
        {  
            id = 0x0B, core = psm,  
            authentication = rsa,  
            pskfile = PSK1.pem,  
            sskfile = SSK1.pem,  
        }  
    }  
}
```

```
        file = psm_fw.elf
    }

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

big_endian

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

```
{ big_endian, file=<partition> }
```

描述

指定二进制文件为大字节序格式。

注释: Bootgen 会自动检测 .elf 文件的字节序。此项仅对二进制文件有效。

实参

指定的分区。

示例

- 对于 Zynq 器件和 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
}
```

- 对于 Versal 自适应 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

语法

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

描述

该属性可指定用于对 bbram 黑密钥进行加密的 IV。bbram_kek_iv 搭配 keysrc=bbram_blk_key 使用时即为有效。

示例

请参阅 [含多个密钥源的 AES 加密示例](#) 以查看示例。

bh_kek_iv

语法

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

描述

该属性可指定用于对启动报头黑密钥进行加密的 IV。bh_kek_iv 搭配 keysrc=bh_blk_key 使用时即为有效。

示例

请参阅 [含多个密钥源的 AES 加密示例](#) 以查看示例。

bh_keyfile

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

```
bh_keyfile = <key file path>
```

描述

将存储在启动报头中的 256 位模糊密钥或黑密钥。仅当加密密钥源为模糊密钥或黑密钥时，此项才有效。

注释：Versal 器件不支持模糊密钥。

实参

模糊密钥或黑密钥路径（基于所选的密钥源）。

示例

- 对于 Zynq 器件和 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
}
```

- 对于 Versal 自适应 SoC:

```
all:
{
    bh_keyfile = bh_key1.txt
    bh_kek_iv = blk_iv.txt
    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
        }
        {
            type=cdo, encryption = aes,
        }
    }
}
```

```
        keysrc = bh_blk_key, aeskeyfile = key3.nky,
        file=fpd_data.cdo
    }
}
```

bh_key_iv

语法

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

描述

解密黑密钥时使用的初始化矢量。

实参

文件路径。

示例

```
Sample BIF - test.bif
all:
{
    [keysr 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

语法

```
[bhsignature] <signature-file>
```

描述

将启动报头签名导入身份验证证书。如果您不愿意共享秘密密钥 PSK，则可使用此项。您可创建签名并将其提供给 Bootgen。

示例

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

blocks

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

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

描述

指定加密中的密钥滚动功能的块大小。每个模块均使用其专用密钥进行加密。初始密钥存储在器件上的密钥源中，而每个后续模块的密钥则在前一个模块中进行加密（封装）。

实参

- <size>: 指定块大小（字节）。

示例

- 对于 AMD Zynq™ UltraScale+™ MPSoC:

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

- 对于 Versal 自适应 SoC:

```
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 = bbram_red_key,  
        aeskeyfile = key1.nky,  
        dpacm_enable,  
        blocks = 8192(20);4096(*),  
        file = lpd_data.cdo  
    }  
    partition  
    {  
        id = 0x0B, core = psm,  
        encryption = aes,  
        keysrc = bbram_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
    }
}
```

注释：在上例中，前 2 个块大小为 4096 字节，后接 1 个大小为 1024 字节的块，再后接 2 个大小为 2048 字节的块。其余块大小均为 4096 字节。

boot_config

语法

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

描述

该属性可指定用于配置启动镜像的参数。选项为：

- `bh_auth_enable`: 启用启动报头身份验证，对启动镜像执行身份验证，同时不执行 PPK 散列和 SPK ID 验证。
- `pufhd_bh`: PUF 帮助程序数据存储在启动报头中（默认值为 efuse）。使用 `puf_file` 选项将 PUF 帮助程序数据文件传递到 Bootgen。
- `puf4kmode`: PUF 调整为在 4k 位特征配置内使用（默认值为 12k 位）。
- `shutter = <value>`: 32 位 PUF_SHUT 寄存器值，用于配置 PUF 的快门偏移时间和快门打开时间。
- `smap_width = <value>`: 定义 SelectMAP (SMAP) 总线宽度。

选项包括：

- 对应单片/主 SLR 的选项包括 8、16 和 32（默认 32 位）
- 0 仅适用于 SSI 技术从 SLR

注释：SSI 技术从 SLR 设为 `smap_width=0`，以指示内部下游连接。该选项值不得更改，且仅适用于 SSI 技术从 SLR。

- `dpacm_enable`: 启用 DPA 对策
- `a_hwrot`: 非对称硬件信任根 (A-HWRoT) 启动模式。Bootgen 会根据设计规则检查所采用的 A-HWRoT 启动模式。仅对量产级 PDI 有效。
- `s_hwrot`: 非对称硬件信任根 (S-HWRoT) 启动模式。Bootgen 会根据设计规则检查所采用的 S-HWRoT 启动模式。仅对量产级 PDI 有效。

示例

```
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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

注释: 在以 BIF 为目标 (对于 Versal, 则以 PDI 为目标) 的主启动镜像中, 需添加该属性。

指定辅助启动器件。指示分区所在的辅助启动器件。

实参

适用于 Zynq 器件和 Zynq UltraScale+ MPSoC 的选项包括:

- qspi32
- qspi24
- nand
- sd0
- sd1
- sd-ls
- mmc
- usb
- ethernet
- pcie
- sata

地址字段用于指定给定闪存器件中镜像的偏移。

对于 emmc/mmc 器件, 地址字段不指示闪存器件上的直接地址。PLM 对 emmc/mmc 地址中的位的解读方式如下:

- 0-15: 文件编号
- 16-19: 逻辑驱动编号/文件系统索引

Versal 自适应 SoC 的选项：

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

示例

注释：以下示例对应主启动镜像的 BIF。

- Zynq 器件和 Zynq UltraScale+ MPSoC 示例如下：

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

- AMD Versal™ 自适应 SoC 示例如下：

```
new_bif:  
{  
    id_code = 0x04ca8093  
    extended_id_code = 0x01  
    id = 0x2  
    boot_device { mmc, address=0x10000 }  
    image  
    {  
        name = pmc_subsys, id = 0x1c000001  
        { id = 0x01, type = bootloader, file = plm.elf }  
        { id = 0x09, type = pmcdtata, load = 0xf2000000, file =  
          pmcdtata.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.rpdo }  
    { id = 0x05, type = cdo, file = system.rnpi }  
}  
image  
{  
    name = fpd, id = 0x420c003  
    { id = 0x08, type = cdo, file = fpd_data.cdo }  
}  
}
```

bootimage

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

用于指定以下文件规格为 Bootgen 所创建的启动镜像，并作为输入来加以复用。

实参

指定的文件名。

示例

- 对于 FSBL:

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

在以上示例中，`fsbl.bin` 和 `system.bin` 均为使用 Bootgen 生成的镜像。

- 对于 fsbl.bin 生成:

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

请使用以下命令:

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

- 对于 system.bin 生成:

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

请使用以下命令:

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

- 对于 AMD Versal™ 自适应 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 }  
    }  
}
```

注释: *base.pdi 即为 Vivado 所生成的 PDI。

bootloader

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

用于将 ELF 文件标识为 FSBL 或 PLM。

- 仅限 ELF 文件才可包含该属性。
- 仅限将 1 个文件指定为启动加载程序。
- 此 ELF 文件的程序报头只能包含一个 filesz > 0 的 LOAD 节，该节必须可执行（必须设置 x 标志）。

实参

指定的文件名。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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 = pmcdtata, load = 0xf2000000, file =  
pmc_data.cdo }  
    }  
}
```

bootvectors

语法

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

描述

该属性用于指定就地执行 (XIP) 的矢量表。

示例

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

checksum

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

用于指定需要验证校验和的分区。更安全的功能（如 [身份验证](#) 和 [加密](#)）不支持该属性。

实参

- none: 不执行校验和操作。
- MD5: 针对 AMD Zynq™ 7000 SoC 器件执行 MD5 校验和操作。在这些器件中，针对启动加载程序不支持执行校验和操作。
- SHA3: 针对 AMD Zynq™ UltraScale+™ MPSoC 器件和 Versal 自适应 SoC 执行校验和操作。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

copy

语法

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

描述

该属性可指定将镜像复制到位于指定地址的存储器。

示例

```
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

语法

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

描述

该属性用于指定执行分区的核。

实参

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

示例

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

注释：*base.pdi 即为 Vivado 所生成的 PDI。

delay_auth

语法

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

描述

该属性表示将于后续阶段执行身份验证。这有助于 Bootgen 在分区加密期间保留空间以供散列使用。

示例

```
stage2b:  
{  
    image  
    {  
        name = lpd  
        id = 0x4210002  
        partition  
        {  
            id = 0x0C,  
            type = cdo,  
            encryption=aes, delay_auth
```

```
    keysrc = bbram_red_key,
    aeskeyfile = lpd_data.nky,
    file = lpd_data.cdo
}
}
```

delay_handoff

语法

```
{ delay_handoff }
```

描述

该属性可指定延迟交接至子系统。

示例

```
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

语法

```
{ delay_load }
```

描述

该属性可指定延迟加载子系统。

示例

```
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

语法

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

描述

指定用于执行分区的核。以下示例指定 FSBL 在 A53-0 核上执行，而应用则将在 R5-0 核上执行。

注释：

- FSBL 只能在 A53-0 或 R5-0 上执行。
- PMU 由 FSBL 加载: [destination_cpu=pmu] pmu.elf，在此流程中，BootROM 首先加载 FSBL，然后 FSBL 加载 PMU 固件。
- PMU 由 BootROM 加载: [pmufw_image] pmu.elf。在此流程中，BootROM 首先加载 PMU，然后加载 FSBL，这样 PMU 即可在 FSBL 启动前执行功耗管理任务。

实参

- a53-0 (默认值)
- a53-1
- a53-2
- a53-3
- r5-0
- r5-1
- r5-lockstep
- pmu

示例

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

destination_device

语法

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

描述

指定分区目标为 PS 还是 PL。

实参

- ps: 分区目标为 PS。这是默认值。
- pl: 分区目标为 PL (适用于比特流)。

示例

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

early_handoff

语法

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

描述

此标志可确保加载分区后立即交接至关键应用；否则，将按顺序先加载所有分区，然后同样按顺序执行交接。

注释：在以下场景中，FSBL 先加载 app1、然后加载 app2，并立即将控制权先交接给 app2 再交接给 app1。

示例

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

efuse_kek_iv

语法

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

描述

该属性可指定用于对 eFUSE 黑密钥进行加密的 IV。“efuse_kek_iv”搭配“keysrc=efuse_blk_key”使用时即为有效。

示例

请参阅[含多个密钥源的 AES 加密示例](#)以查看示例。

efuse_user_kek0_iv

语法

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

描述

该属性可指定用于对 eFUSE 用户黑密钥 key0 进行加密的 IV。“efuse_user_kek0_iv”搭配“keysrc=efuse_user_blk_key0”使用时即为有效。

示例

请参阅[含多个密钥源的 AES 加密示例](#)以查看示例。

efuse_user_kek1_iv

语法

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

描述

该属性可指定用于对 eFUSE 用户黑密钥 key1 进行加密的 IV。“efuse_user_kek1_iv”搭配“keysrc=efuse_user_blk_key1”使用时即为有效。

示例

请参阅[含多个密钥源的 AES 加密示例](#)以查看示例。

encryption

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

用于指定需要进行加密的分区。加密算法为：

实参

- none：分区不执行加密。这是默认值。
- aes：分区使用 AES 算法执行加密。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

    metaheader
    {
        encryption = aes,
        keysrc = bbram_red_key,
```

```
        aeskeyfile = efuse_red_metaheader_key.nky,
    }

image
{
    name = pmc_subsys, id = 0x1c000001
    partition
    {
        id = 0x01, type = bootloader,
        encryption = aes,
        keysrc = bbram_red_key,
        aeskeyfile = bbram_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 = bbram_red_key,
        aeskeyfile = key1.nky,
        file = lpd_data.cdo
    }
    partition
    {
        id = 0x0B, core = psm,
        encryption = aes,
        keysrc = bbram_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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

核必须配置为所示异常级别。

实参

- el-0
- el-1
- el-2
- el-3 (默认值)

示例

- 对于 Zynq 器件和 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
}
```

- 对于 AMD Versal™ 自适应 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 }
    }
}
```

注释: *base.pdi 即为 Vivado 所生成的 PDI。

familykey

语法

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

描述

指定族密钥。要获取族密钥，请通过 secure.solutions@xilinx.com 联系 AMD 代表。

实参

文件路径。

示例

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

file

语法

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

描述

该属性可指定用于创建分区的文件。

示例

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

注释：*base.pdi 即为 Vivado 所生成的 PDI。

fsbl_config

语法

```
[fsbl_config <options>] <partition>
```

描述

该选项可指定用于配置启动镜像的参数。FSBL，应在 A53 上以 64 位启动报头身份验证模式运行。

实参

- `bh_auth_enable`: 启用启动报头身份验证：对启动镜像执行 RSA 身份验证时，不执行 PPK 散列和 SPK ID 验证。
- `auth_only`: 启动镜像仅使用 RSA 签名。FSBL 不应进行解密。如需了解更多信息，请参阅《Zynq UltraScale+ 器件技术参考手册》([UG1085](#))。
- `opt_key`: 使用运行密钥进行 block-0 解密。安全报头具有运行密钥。
- `pufhd_bh`: PUF 帮助程序数据存储在启动报头中（默认值为 `efuse`）。使用 `[puf_file]` 选项将 PUF 帮助程序数据文件传递到 Bootgen。
- `puf4kmode`: PUF 调整为在 4k 位配置内使用（默认值为 12k 位）。
- `shutter = <value>`: 32 位 `PUF_SHUT` 寄存器值，用于配置 PUF 的快门偏移时间和快门打开时间。

注释：此快门值必须与 PUF 寄存期间使用的快门值相匹配。

示例

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

headersignature

语法

对于 Zynq UltraScale+ MPSoC:

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

对于 Versal 自适应 SoC:

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

描述

将报头签名导入身份验证证书。如果您不想共享秘密密钥，则可使用此项。您可创建签名并将其提供给 Bootgen。

实参

```
<signature_file>
```

示例

对于 Zynq UltraScale+ MPSoC:

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

对于 Versal 自适应 SoC:

```
stage5:
{
    bhsignature = botheader.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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

```
{ hivec, file=<partition> }
```

描述

用于将“Exception Vector Table”（异常矢量表）的位置指定为 `hivec`。这仅适用于 a53 核（32 位）与 r5 核。

- `hivec`: 异常矢量表位于 0xFFFF0000。
- `lavec`: 异常矢量表位于 0x00000000。这是默认值。

实参

无

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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

语法

```
id = <id>
```

描述

该属性可基于其定义位置来指定以下 ID：

- pdi ID - 在最外层/PDI 括号内
- 镜像 ID - 在镜像括号内
- 分区 ID - 在分区括号内

给定镜像的镜像 ID 是固定的。请参阅下表以获取 AMD 为 Versal 自适应 SoC 定义的镜像 ID。

表 39：镜像 ID（对于给定分区是固定的）

镜像	子系统/域	镜像 ID 值	描述
PMC	子系统	0x1C000001	PMC 子系统 ID
PLD	域	0x18700000	PLD0 器件 ID (因为 PLD0 表示整个 PLD 域)
LPD	域	0x04210002	LPD 电源节点 ID
FPD	域	0x0420C003	FPD 电源节点 ID
默认子系统	子系统	0x1C000000	默认子系统 ID
CPD	域	0x04218007	CPM 电源节点 ID
AIE	域	0x0421C005	AIE 电源节点 ID

注释：对于 PS 分区（例如 A72 和 R5 ELF），请使用默认子系统 ID。

注释：分区 ID 用于标识分区，不可用于 PLM 处理。Partition ID（分区 ID）可由您根据自己的编号方案进行更改。PDI ID 和镜像 ID 不应更改。

示例

```
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

语法

```
image
{
```

描述

该属性用于定义子系统/镜像。

示例

```
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

语法

```
imagestore = <id>
```

描述

用于指定要添加到“Image Store”（镜像存储）的 PDI 的 ID。“Image Store”功能特性允许将 PDI 文件存储在存储器 (DDR) 中，稍后用于加载 PDI 内的指定镜像。其目的是为了允许无需依赖外部启动器件即可执行部分重配置、子系统重启等操作。

示例

```
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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

```
init = <filename>
```

描述

位于启动加载程序末尾的寄存器初始化块，通过解析 .int 文件规格来构建。允许最多 256 个地址/值对。.int 文件具有特定格式。

示例

BIF 文件样本如下所示：

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

keysrc

语法

```
keysrc = <options>
```

描述

用于指定加密的密钥源。

实参

启动加载程序、Meta 报头和分区的有效密钥源为：

- efuse_red_key
- efuse_blk_key
- bbram_red_key
- bbram_blk_key
- bh_blk_key

还有些仅针对分区有效的密钥源：

- user_key0
- user_key1
- user_key2
- user_key3
- user_key4
- user_key5
- user_key6
- user_key7
- efuse_user_key0
- efuse_user_blk_key0

示例

```
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

语法

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

描述

用于指定加密的密钥源。

实参

- bbram_red_key: BBRAM 中存储的红密钥
- efuse_red_key: eFUSE 中存储的红密钥
- efuse_gry_key: eFUSE 中存储的灰（模糊）密钥。
- bh_gry_key: 启动报头中存储的灰（模糊）密钥。
- bh_blk_key: 启动报头中存储的黑密钥。
- efuse_blk_key: eFUSE 中存储的黑密钥。
- kup_key: 用户密钥。

示例

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

FSBL 是使用 encr.nky 密钥加密的，此密钥存储在 eFUSE 中用于解密。

load

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

为存储器内的分区设置加载地址。

示例

- 对于 Zynq 器件和 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  
}
```

- 对于 AMD Versal™ 自适应 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 }  
    }  
}
```

注释: *base.pdi 即为 Vivado 所生成的 PDI。

metaheader

语法

```
metaheader { }
```

描述

注释: metaheader 支持所有安全性属性。

该属性用于定义 Meta 报头的加密和身份验证属性，例如，密钥、密钥源等。

示例

```
test:  
{  
    metaheader  
    {  
        encryption = aes,  
        keysrc = bbram_red_key,  
        aeskeyfile = headerkey.nky,  
        authentication = rsa  
    }  
    image  
    {  
        name = pmc_subsys, id = 0x1c000001  
        {  
            type = bootloader,  
            encryption = aes,  
            keysrc = bbram_red_key,  
            aeskeyfile = key1.nky,  
            blocks = 8192(*),  
            file = plm.elf  
        }  
        {  
            type=pmcdata,  
            load=0xf2000000,  
            aeskeyfile=key2.nky,  
            file=pmc_cdo.bin  
        }  
    }  
}
```

name

语法

```
name = <name>
```

描述

该属性可指定镜像/子系统的名称。

示例

```
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_firmware.elf }  
}  
image  
{  
    name = pl_cfi, id = 0x18700000  
    { id = 0x03, type = cdo, file = system.rcd0 }  
    { id = 0x05, type = cdo, file = system.rnpi }  
}  
image  
{  
    name = fpd, id = 0x420c003  
    { id = 0x08, type = cdo, file = fpd_data.cdo }  
}  
}
```

offset

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

用于设置启动镜像中的分区的绝对偏移。

实参

指定的值和分区。

示例

- 对于 Zynq 器件和 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  
}
```

- 对于 AMD Versal™ 自适应 SoC:

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

注释: *base.pdi 即为 Vivado 所生成的 PDI。

optionaldata

语法

```
optionaldata {<filename>, id=<id>}
```

描述

该属性允许您指定 ID 和数据文件。在 BIF 中也可以指定多个 optionaldata。数据文件必须为含扩展名 .bin 的二进制文件。每个可选数据文件都需要唯一 ID，用于识别相关的可选数据。对于可选数据，从 0x0 至 0x20 之间的 ID 均保留供内部使用。用户可选数据 ID 可采用 > 0x20 的任何 ID。在 PDI 中，该可选数据排在 IHT 之后。

实参

文件名

id

示例

```
new_bif:  
{  
    id_code = 0x04ca8093  
    extended_id_code = 0x01  
    id = 0x2  
  
    optionaldata {data2.bin, id=33}  
    optionaldata {data3.bin, id=34}  
  
    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}  
    }  
}
```

overlay_cdo

语法

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

描述

配合 overlay_cdo 命令一起使用的输入文件将包含需覆盖的标记和内容。Bootgen 会在 BIF 中存在的所有 CDO 文件中搜索类似标记，找到其中内容后，会将 CDO 替换为来自覆盖 CDO 的内容。

parent_id

语法

```
parent_id = <id>
```

描述

该属性可指定父 PDI 的 ID。用于识别部分 PDI 与其对应启动 PDI 之间的关系。

示例

```
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

语法

```
partition  
{  
}
```

描述

该属性用于定义分区。这是可选属性，用于简化 BIF 并使其可读。

示例

```
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  
        }  
    }  
}
```

注释：分区属性为可选，并且可编写无属性的 BIF 文件。

编写以上 BIF 时，其中可不含分区属性，如下所示：

```
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 和 owner

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

负责加载分区的分区的所有者。

实参

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:
 - fsbl: 由 FSBL 加载此分区
 - uboot: 由 U-Boot 加载此分区
- 对于 AMD Versal™ 自适应 SoC:
 - plm: 由 PLM 加载此分区
 - non-plm: PLM 忽略此分区，并以其他方式加载此分区

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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

语法

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

描述

指定分区 ID。默认值为 0。

注释：如未指定 PID，则此 ID 会在标准流程中的每个分区上递增。如在 HSM 流程中不指定 PID，那么由于每个分区都单独处理，因此 PID 始终保持为 0。此行为会导致最终镜像之间出现不匹配。

示例

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

pmufw_image

语法

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

描述

PMU 固件镜像将由 BootROM 加载（在加载 FSBL 前）。适用于 pmufw_image 的选项将随启动加载程序分区直接插入。Bootgen 不考虑随 pmufw_image 选项与其提供的任何附加属性。

实参

文件名

示例

```
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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

```
ppkfile = <filename>
```

描述

PPK 密钥用于对启动镜像中的分区进行身份验证。

请参阅 [使用身份验证](#)。

实参

指定的文件名。

注释: 秘密密钥文件包含密钥的公钥组件。只要注明 PSK，您就无需指定 PPK。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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

语法

对于 Zynq 7000 和 Zynq UltraScale+ MPSoC 器件：

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

对于 Versal 自适应 SoC：

```
presign = <signature_file>
```

描述

将分区签名导入分区身份验证证书。如果您不愿意共享秘密密钥 (SSK)，则可使用此项。您可创建签名并将其提供给 Bootgen。

- <signature_file>：指定签名文件。
- <partition>：监听 <signature_file> 应用到的分区。

示例

对于 Zynq 7000 和 Zynq UltraScale+ MPSoC 器件：

```
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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC：

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

- 对于 AMD Versal™ 自适应 SoC：

```
pskfile = <filename>
```

描述

此 PSK 用于对启动镜像中的分区进行身份验证。如需了解更多信息，请参阅 [使用身份验证](#)。

实参

指定的文件名。

注释：秘密密钥文件包含密钥的公钥组件。只要注明 PSK，您就无需指定 PPK。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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 = pmcdt, load = 0xf2000000, file=pmc_cdo.bin }  
        { type=cdo, authentication=rsa, file=fpd_cdo.bin,  
          pskfile = primary1.pem, sskfile = secondary1.pem }  
    }  
}
```

puf_file

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 Versal 自适应 SoC:

```
puf_file = <puf data file>
```

描述

PUF 帮助程序数据文件。

- PUF 配合黑密钥用作为加密密钥源。
- PUF 帮助程序数据大小为 1544 字节。
- 其中 1536 字节用于 PUF HD + 4 字节用于 CHASH + 3 字节用于 AUX + 1 字节用于对齐。

如需了解更多信息，请参阅 [黑密钥/PUF 密钥](#)。

示例

- 对于 Zynq 器件和 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  
}
```

- 对于 AMD Versal™ 自适应 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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

该属性会为特定分区保留存储器。即使分区大小小于保留的存储器，分区长度仍始终设为保留的大小。如果分区大小大于保留的大小，那么分区长度是分区的实际大小。

如果您要更新启动镜像中的分区，而不更改对应的报头，那么该属性很有用。

实参

指定的分区

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

注释: *base.pdi 即为 Vivado 所生成的 PDI。

split

语法

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

描述

根据模式将镜像拆分为多个部分。Slaveboot 模式按如下方式拆分：

- 启动报头 + 启动加载程序
- 镜像和分区报头
- 其他分区

Normal 模式按如下方式拆分：

- 启动报头 + 镜像报头 + 分区报头 + 启动加载程序
- Partition1
- Partition2 以此类推

仅限 Zynq UltraScale+ MPSoC 才支持 Slaveboot，Zynq 7000 和 Zynq UltraScale+ MPSoC 均支持 normal。除拆分模式外，输出格式还可指定为 bin 或 mcs。

选项

实参模式可用选项为：

- slaveboot
- normal
- bin
- mcs

示例

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

注释：选项拆分模式 normal 与命令行选项 split 相同。按计划将弃用此命令行选项。

注释：Versal 自适应 SoC 不支持 Split slaveboot 模式。

spkfile

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

```
spkfile = <filename>
```

描述

SPK 用于对启动镜像中的分区进行身份验证。如需了解更多信息，请参阅 [使用身份验证](#)。

实参

指定的文件名。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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 = pmcdtdata, load = 0xf2000000, file=pmc_cdo.bin }  
        { type=cdo, authentication=rsa, file=fpd_cdo.bin}  
          spkfile=secondary1.pub, sskfile = secondary1.pem }  
    }  
}
```

注释: 密密钥文件包含密钥的公钥组件。只要注明 SSK，您就无需指定 SPK。

spksignature

语法

对于 Zynq 和 Zynq UltraScale+ MPSoC 器件:

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

对于 Versal 自适应 SoC:

```
spksignature = <signature file>
```

描述

将 SPK 签名导入身份验证证书。您不愿意共享秘密密钥 PSK 时可使用此项，您可以创建签名并将其提供给 Bootgen。

实参

指定的文件名。

示例

对于 Zynq 和 Zynq UltraScale+ MPSoC 器件：

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

对于 Versal 自适应 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

语法

```
[spk_select = <options>]
```

或

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

描述

选项包括：

- spk_efuse：指示针对该分区使用 spk_id eFUSE。这是默认值。
- user_efuse：指示针对该分区使用用户 eFUSE。

由 CSU ROM 加载的分区始终使用 spk_efuse。

注释：spk_id eFUSE 可指定有效的密钥。因此，ROM 会根据 SPK ID 检查 spk_id eFUSE 的整个字段，以确保其位对位匹配。

用户 eFUSE 可指定无效（已撤销）的密钥 ID。因此，固件（非 ROM）会检查表示 SPK ID 的给定用户 eFUSE 是否已烧录。`spk_select = user-efuse` 指示针对该分区使用用户 eFUSE。

示例

```
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

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

```
sskfile = <filename>
```

描述

SSK 用于对启动镜像中的分区进行身份验证。如需了解更多信息，请参阅 [使用身份验证](#)。

实参

指定的文件名。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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 = pmcd, load = 0xf2000000, file=pmc_cdo.bin }  
        { type=cdo, authentication=rsa, file=fpd_cdo.bin, pskfile =  
          primary1.pem, sskfile = secondary1.pem }  
    }  
}
```

注释: 密钥文件包含密钥的公钥组件。只要注明 PSK，您就无需指定 PPK。

startup

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

该选项用于在加载分区后，设置其输入地址。针对不执行的分区忽略此项。此项仅对二进制分区有效。

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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 }  
    }  
}
```

注释: *base.pdi 即为 Vivado 所生成的 PDI。

trustzone

语法

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 SoC:

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

描述

将核配置为安全或非安全 TrustZone。选项包括:

- secure
- nonsecure (默认值)

示例

- 对于 Zynq 器件和 Zynq UltraScale+ MPSoC:

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

- 对于 AMD Versal™ 自适应 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 }  
    }  
}
```

```
{  
    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 }  
}
```

注释: *base.pdi 即为 Vivado 所生成的 PDI。

type

语法

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

描述

该属性用于指定分区类型。选项如下。

- bootloader
- pmcdata
- cdo
- bootimage

示例

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

注释: *base.pdi 即为 Vivado 所生成的 PDI。

udf_bh

语法

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

描述

将要复制的数据文件导入启动报头的用户定义字段 (UDF)。用户定义的输入数据通过字符串格式的文本文件来提供。AMD SoC 的 UDF 中字节总数为：

- zynq: 76 字节
- zynqmp: 40 字节

实参

指定的文件名。

示例

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

以下是 udf_bh 的输入文件示例：

udf_bh - test.txt 的输入文件样本

```
123456789abcdef85072696e636530300301440408706d616c6c6164000508  
266431530102030405060708090a0b0c0d0e0f101112131415161718191a1b  
1c1d1
```

udf_data

语法

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

描述

将包含最多 56 字节数据的文件导入身份验证证书的用户定义字段 (UDF)。如需了解更多信息，请参阅 [身份验证](#) 以了解有关身份验证证书的更多信息。

实参

指定的文件名。

示例

```
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

语法

```
userkeys = <filename>
```

文件格式

```
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>
```

描述

指向用户密钥文件的路径。密钥文件包含用于加密分区的用户密钥。用户密钥的大小可以为 128 或 256 位。128 位密钥只能用于运行时加载的分区。

注释：在 SSI 技术器件中，不支持 userkeys

示例

在以下示例中，FPD 分区将密钥源用作 user_key2，因此该分区的 .nky 文件必须使用 userkeys 文件中的 user_key2 作为 key0。然后，Bootgen 使用 .nky 文件中的此 key0 进行加密。解密期间，PLM 将使用由 pmc_data 编程的 user_key2。

```
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

语法

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

描述

指示直接从 QSPI 闪存执行 FSBL 的“eXecute In Place”（就地执行）。

注释：该属性仅适用于 FSBL/启动加载程序分区。

实参

指定的分区。

示例

此示例显示如何针对 AMD Zynq™ UltraScale+™ MPSoC 器件创建就地执行的启动镜像。

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

命令参考

如需了解其中每个命令支持的器件家族，请参阅 [命令和描述](#)。

arch

语法

```
-arch [options]
```

描述

需为其创建启动镜像的 AMD 家族架构。

实参

- zynq：AMD Zynq™ 7000 器件架构。这是默认值，对应需为其创建启动镜像的家族架构。
- Zynqmp：AMD Zynq™ UltraScale+™ MPSoC 架构。
- fpga：镜像以其他 FPGA 架构为目标。
- versal：此镜像目标为 AMD Versal™ 器件。

返回值

无

示例

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

authenticatedjtag

语法

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

描述

用于在安全启动期间启用 JTAG。

实参

- rsa
- ecdsa

示例

```
bootgen -arch versal -image boot.bif -w -o boot.bin -authenticatedjtag rsa  
authJtag-rsa.bin
```

bif_help

语法

```
bootgen -bif_help
```

```
bootgen -bif_help aeskeyfile
```

描述

列出受支持的 BIF 文件属性。如需获取每个 bif 属性的详细说明，请在命令行上将属性名称指定为 -bif_help 的实参。

dual_ospis_mode

语法

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

描述

对于双 OSPI 堆叠配置生成 2 个输出文件，需注明闪存大小（以 MB 为单位）（64、128 或 256）。

示例

此示例可生成 2 个输出文件用于对双 OSPI 堆叠配置中的 2 个闪存进行独立编程。实际镜像的前 64 MB 将写入第一个文件，剩余部分将写入第二个文件。如果实际镜像本身不足 64 MB，则仅生成 1 个文件。仅限 Versal 自适应 SoC 才支持该选项。

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

实参

- stacked, <size>

dual_qspi_mode

语法

```
bootgen -dual_qspi_mode [parallel] | [stacked <size>]
```

描述

生成 2 个输出文件用于双 QSPI 配置。对于堆叠配置，需注明闪存大小（以 MB 为单位）（16、32、64、128 或 256）。

示例

此示例可生成 2 个输出文件用于对双 QSPI 并行配置中的 2 个闪存进行独立编程。

```
bootgen -image test.bif -o -boot.bin -dual_qspi_mode parallel
```

此示例可生成 2 个输出文件用于对双 QSPI 堆叠配置中的 2 个闪存进行独立编程。实际镜像的前 64 MB 将写入第一个文件，剩余部分将写入第二个文件。如果实际镜像本身不足 64 MB，则仅生成 1 个文件。

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

实参

- parallel
- stacked <size>

dump

语法

```
-dump <filename> [arguments]
```

描述

该选项用于从 PDI 转储不同组件。对于 SSI 技术器件，该选项可用于将中间从 PDI 转储为二进制文件。此外，它还可在生产 PDI 时直接提取启动报头内容。以下演示了一个示例。

示例

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

```
bootgen -arch versal -dump boot.pdi bh
```

文件名

PDI 文件

实参

- empty：将分区转储为二进制文件。
- bh：将启动报头转储为的独立文件。
- boot_files：转储所有启动文件（启动报头、PLM 和 PMC CDO）
- plm：转储 plm 分区
- pmc_cdo：转储 pmc_cdo 分区
- slave_pdis：转储从 PDI 以供 SSI 技术器件使用

注释：启动报头将随 PDI 一起转储为独立二进制文件。生成的 PDI 不会从启动报头中剥离，但保留启动报头。

注释：如不指定任何实参，则转储所有分区。

dump_dir

语法

```
dump_dir <path>
```

描述

该选项用于指定目录位置，以便写入 -dump 命令内容。

示例

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

efuseppkbits

语法

```
bootgen -image test.bif -o boot.bin -efuseppkbits efusefile.txt
```

实参

efusefile.txt

描述

该选项可指定要写入的 eFUSE 文件（其中包含 PPK 散列）的名称。该选项可生成不含任何填充的直接散列。生成的 efusefile.txt 文件包含 PPK 密钥散列，其中：

- AMD Zynq™ 7000 使用 SHA2 协议来处理散列。
- AMD Zynq™ UltraScale+™ MPSoC 和 Versal 自适应 SoC 使用 SHA3 来处理散列。

enable_auth_opt

语法

```
bootgen -arch versal -image test.bif -o boot.bin -enable_auth_opt
```

描述

该选项用于启用身份验证优化。生成的启动镜像含 Meta 报头和分区散列，存储在可选数据内。

注释：仅限 AMD Versal™ SoC 才支持该选项。

encrypt

语法

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

描述

该选项用于指定加密执行方式以及密钥存储位置。NKY 密钥文件通过 BIF 文件属性 aeskeyfile 来传递。使用命令行时仅指定密钥源。

实参

密钥源实参：

- efuse：AES 密钥存储在 eFUSE 中。这是默认值。
- bbram：AES 密钥存储在 BBRAM 中。

encryption_dump

语法

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

描述

生成加密 log 日志文件 aes_log.txt。生成的 aes_log.txt 包含用于对每个数据块进行加密的 AES Key/IV 对的详细信息。它还会记录分区以及用于对分区进行加密的 AES 密钥文件。

注释：仅限 AMD Zynq™ UltraScale+™ MPSoC 才支持该选项。

示例

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

fill

语法

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

描述

该选项可指定用于按 <hex byte> 格式对填充/保留存储器进行填充的字节。

输出

0xAB 字节中的 boot.bin 文件。

示例

生成的输出镜像名为 boot.bin。输出镜像的格式是根据随 -o 选项提供的文件的文件扩展名来判定的，其中 -fill：用于指定要填充的字节。<hex byte> 在报头表（而不是 0xFF）中进行填充。

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

generate_hashes

语法

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

描述

该选项用于为所有分区和要签名的其他组件（如启动报头、镜像报头和分区报头）生成散列文件。该选项可按 AMD Zynq™ 7000 格式生成包含 PKCS#1v1.5 填充散列的文件：

表 40: Zynq: SHA-2 (256 字节)

值	SHA-2 Hash	T-Padding	0x0	0xFF	0x01	0x00
字节数	32	19	1	202	1	1

该选项可按 AMD Zynq™ UltraScale+™ MPSoC 格式生成包含 PKCS#1v1.5 填充散列的文件：

表 41: ZynqMP: SHA-3 (384 位)

值	0x0	0x1	0xFF	0xFF	T-Padding	SHA-3 Hash
字节数	1	1	314	1	19	48

示例

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

Bootgen 会使用指定 BIF 生成以下散列文件：

- 启动报头散列
- spk 散列
- 报头表散列
- fsbl.elf 分区散列
- hello.elf 分区散列

generate_keys

语法

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

描述

该选项可生成用于身份验证的密钥以及用于加密的模糊密钥。

注释：如需了解有关生成加密密钥的更多信息，请参阅 [密钥生成](#)。

身份验证密钥生成示例

身份验证密钥生成示例。此示例可在 BIF 文件指定的路径中生成身份验证密钥。

示例

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

模糊密钥生成示例

此示例可在 familykey.txt 所在路径中生成模糊密钥。

命令：

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

BIF 文件样本如以下示例所示：

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

实参

- rsa
- pem
- obfuscated

h 和 help

语法

```
bootgen -help  
bootgen -help arch
```

描述

列出受支持的命令行属性。如需获取每个属性的详细说明，请在命令行上将属性名称指定为 `-help` 的实参。

image

语法

```
-image <BIFF_filename>
```

描述

该选项用于指定输入 BIF 文件名。BIF 文件用于指定启动镜像的每个组件（按启动顺序）并允许对每个镜像组件指定可选属性。通常每个镜像组件映射到一个分区，但在某些情况下，如果镜像组件在存储器中不连续，即可映射到多个分区。

实参

bif_filename

示例

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

BIF 文件样本如以下示例所示：

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

log

语法

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

描述

生成启动镜像时生成 log 日志。有多个选项可用于选择信息级别。这些信息显示在控制台上以及 log 日志文件中，在当前工作目录中会生成指定的 bootgen_log.txt。

实参

- error: 仅采集错误信息。
- warning: 采集警告信息和错误信息。这是默认值。
- info: 采集常规信息以及上述所有信息。
- trace: 随上述信息一起采集更详细的信息。

nonbooting

语法

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

描述

该选项用于创建中间启动镜像。将生成中间 test.bin 镜像作为输出，即使缺少生成经身份验证的镜像所需的秘密密钥时也是如此。此中间镜像无法启动。

示例

```
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**语法**

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

描述

该选项用于指定输出镜像文件 (.bin 或 .mcs 扩展名) 的名称。

输出

BIN 格式或 MCS 格式的完整启动镜像文件。

示例

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

启动镜像以 MCS 格式输出。

p**语法**

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

描述

该选项用于指定 AMD 器件的部件名称。此名称是生成加密密钥所必需的。它将被原封不变地复制到 *.nky 文件中的 Device 行内。仅当启用加密时，才适用该选项。如果 BIF 文件中指定的路径中不存在密钥文件，那么将在相同路径内生成新加密密钥，并随 xc7z020clg484 一起复制到 nky 文件的 Device 字段中。生成的镜像为加密镜像。

padimageheader

语法

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

描述

该选项可填充镜像报头表和分区报头表，直至达到允许的最大分区数量为止，以强制对齐以下分区。默认情况下，该功能处于已启用状态。指定 0 会禁用该功能。`boot.bin` 包含实际数量的镜像报头表和分区报头表，无需填充额外的表。如果未指定任何选项或者如果 `-padimageheader=1`，那么将填充全部镜像报头表和分区报头表，直至达到最大分区数量为止。

实参

- 1: 填充报头表，直至最大分区数为止。这是默认值。
- 0: 不填充报头表。

镜像或分区报头长度

- 对于 Zynq 器件，允许的最大分区数量为 14。
- 对于 Zynq UltraScale+ MPSoC，允许的最大分区数量为 32。

process_bitstream

语法

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

描述

仅处理来自 BIF 的比特流，并将其作为 MCS 文件或 BIN 文件来输出。例如：如果针对 BIF 文件中的比特流选中加密，那么输出即为已加密的比特流。

实参

- bin: 以 BIN 格式输出。
- mcs: 以 MCS 格式输出。

返回

生成的输出为 BIN 格式或 MCS 格式的比特流；随附经过处理的文件（不含任何报头）。

read

语法

```
-read <filename> [options]
```

描述

用于根据选项读取启动报头、镜像报头和分区报头。它还会转储用户可选数据。

实参

- bh: 从启动镜像读取人工可读格式的启动报头
- iht: 从启动镜像读取镜像报头表
- ih: 从启动镜像读取镜像报头
- pht: 从启动镜像读取分区报头
- ac: 从启动镜像读取身份验证证书

示例

```
bootgen -arch zynqmp -read BOOT.bin
```

```
bootgen -arch versal -read BOOT.bin
```

注释: MCS 格式下的启动镜像/PDI 不支持 read

spksignature

语法

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

描述

该选项用于生成 SPK 签名文件。仅当在 BIF 中指定 `spkfile` 和 `pskfile` 时，才必须使用该属性。将生成 SPK 签名文件 (`spksignfile.txt`)。

选项

指定要生成的签名文件的名称。

split

语法

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

描述

该选项可将含报头的每个数据分区作为一个新文件 (MCS 格式或 BIN 格式) 来输出。

输出

生成的输出文件为：

- 启动报头 + 镜像报头 + 分区报头 + Fsbl.elf
- Partition1.bit
- Partition2.elf

示例

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

verify

语法

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

描述

该选项用于对启动镜像的身份验证执行验证。根据可用分区对启动镜像中的所有身份验证证书进行验证。通过下列步骤来执行验证：

1. 验证报头身份验证证书：
 - 对于 Zynq UltraScale+ MPSoC：验证 SPK 签名并验证报头签名。
 - 对于 Versal：验证 SPK 签名、验证 IHT 签名并验证 Meta 报头签名。
2. 验证启动加载程序的身份验证证书：验证启动报头签名、验证 SPK 签名并验证启动加载程序签名。
3. 验证分区的身份验证证书：验证 SPK 签名并验证分区签名。

针对给定启动镜像中的所有分区重复此过程。

verify_kdf

语法

```
bootgen -arch zynqmp -verify_kdf testVec.txt
```

描述

testVec.txt 文件格式如下所示。

```
L = 256
KI = d54b6fd94f7cf98fd955517f937e9927f9536caeb148fba1818c1ba46bba3a4
FixedInputDataByteLen = 60
FixedInputData =
94c4a0c69526196c1377cebf0a2ae0fb4b57797c61bea8eeb0518ca08652d14a5e1bd1b116b1
794ac8a476acbdbbcd4f6142d7b8515bad09ec72f7af
```

Bootgen 使用“Counter Mode KDF”（计数器模式密钥衍生函数），根据测试矢量文件中给定的输入数据生成输出密钥 (KO)。此 KO 会打印在控制台上以供您比对。

W

语法

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

描述

该选项用于指定是否覆盖现有文件。如果路径中已存在 boot.bin 文件，则将覆盖此文件。-w on 选项与 -w 选项的处理方式相同。如果未指定 -w 选项，则默认不覆盖此文件。

实参

- on: 随 -w on 命令指定（含实参），或者随 -w 指定（不含实参）。这是默认值。
- off: 指定不覆盖现有文件。

zynqmpes1

语法

```
bootgen -arch zynqmp -image test.bif -o boot.bin -zynqmpes1
```

描述

该选项用于指定生成的镜像用于 ES1 (1.0)。该选项仅在生成经过身份验证的镜像时才有效，否则将被忽略。默认填充方案适用于 (2.0) ES2 及更高版本。

初始化对和 INT 文件属性

初始化对支持您为 MIO 多路复用器和闪存时钟轻松完成处理器系统 (PS) 寄存器的初始化工作。这样即可在将 FSBL 镜像复制到 OCM 之前或者利用就地执行 (XIP) 从闪存执行此镜像前完成 MIO 多路复用器的完整配置，并且支持将闪存器件时钟设置为最大带宽速度。

在启动镜像报头固定部分末尾包含 256 个初始化对。初始化对采用这种设计方式是因为每个初始化对均包含 1 个 32 位地址值和 1 个 32 位数据值。如果不执行初始化，那么所有地址值均包含 0xFFFFFFF，而数据值则包含 0x00000000。默认情况下设置初始化对时使用的文本文件使用 .int 文件扩展名，但此文本文件可采用任意文件扩展名。

在文件名前追加的 [init] 文件属性用于将其识别为 BIF 文件中的 INIT 文件。数据格式包含操作指令后接：

- 1 个地址值
- 1 个 = 字符
- 1 个数据值

该行以分号 (;) 结尾。这是一个.set. 操作指令；例如：

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

Bootgen 可从 INT 文件填充启动报头初始化，上限为 256 对。运行 BootROM 时，它会查看地址值。如果该值并非 0xFFFFFFF，那么 BootROM 会使用该地址值后的下一个 32 位值来写入地址值。BootROM 会循环初始化对并设置值直至它遇到 0xFFFFFFF 地址或者达到第 256 个初始化对。

Bootgen 可提供包含以下运算符的完整表达式求值程序（包含嵌套括号用于强制执行计算优先顺序）：

```
* = multiply/  
= divide  
% = mod  
an address value  
ulo divide  
+ = addition  
- = subtraction  
~ = negation  
>> = shift right  
<< = shift left  
& = binary and  
= binary or  
^ = binary nor
```

数值可采用十六进制 (0x)、八进制 (0o) 或十进制数字。数值表达式保留为 128 位顶点整数。您可在任意表达式运算符周围添加空格以提高可读性。

CDO 实用工具

CDO 实用工具 (cdoutil) 是支持通过各种方式来处理 CDO 文件的程序。CDO 文件为根据时钟、PLL 和 MIO 的用户配置在 AMD Vivado™ Design Suite 中针对 AMD Versal™ 器件创建的二进制文件。CDO 属于 PDI 的一部分，并且由 PLM 加载/执行。对于 AMD Zynq™ 7000 器件和 AMD Zynq™ UltraScale+™ MPSoC，此配置包含在随 FSBL 一起编译的 `ps7/psu_init.c/h` 文件内。

访问

`cdoutil` 包含在 Vivado Design Suite/AMD Vitis™ 统一软件平台/Bootgen 安装内，位置为 `<INSTALL_DIR>/bin/cdoutil`。

用法

`cdoutil` 的常规命令行语法为：

```
cdoutil <options> <input(s)>
```

`cdoutil` 的默认函数是对输入文件解码并打印输出 CDO。

命令行选项

有多个选项可用于更改默认行为：

表 42：命令行选项

选项	描述
<code>-address-filter-file <path></code>	指定地址筛选文件
<code>-annotate</code>	为源输出添加注解，填写命令详细信息
<code>-device <type></code>	指定器件名称，默认为 xcvc1902
<code>-help</code>	打印帮助信息
<code>-output-binary-be</code>	以大字节序二进制格式输出 CDO 命令
<code>-output-binary-le</code>	以小字节序二进制格式输出 CDO 命令
<code>-output-file <path></code>	指定输出文件，默认为 <code>stdout</code>
<code>-output-modules</code>	输出供输入文件使用的模块列表
<code>-output-raw-be</code>	以大字节序原始格式输出 CDO 命令

表 42: 命令行选项 (续)

选项	描述
-output-raw-le	以小字节序原始格式输出 CDO 命令
-output-source	以源格式输出 CDO 命令 (默认)
-remove-comments	移除输入中的内容
-rewrite-block	将块写入命令重新写入多条写入命令
-rewrite-sequential	将顺序写入命令重新写入单一块写入命令
-verbose	打印 log 日志信息
post-process <mode>	对 PMCFW 命令进行后处理，将其转为 PLM 命令
cfu-stream-keyhole-size <size>	覆盖默认 CFU 串流锁眼大小
random-commands <count>	生成 <count> 条随机命令
apropos <keywords>	搜索器件寄存器信息，查找 <keywords> 和输出匹配
overlay <path>	指定覆盖文件

注释: 首选 -output-raw-be，因为 Vivado Design Suite 会以大字节序原始格式生成 CDO。-output-raw-le、-output-binary-be 和 -output-binary-le 均不属于首选选项。

地址筛选文件

地址筛选文件是使用 `-address-filter-file <path>` 指定的。此文件用于指定必须从配置中移除的模块。地址筛选文件为文本文件，其中以连字符（减号）开头的每一行都会指定地址范围，此范围内的所有初始化都应移除。示例：

```
# Remove configuration of UART0
-UART0
```

可使用 `-output-modules` 选项生成设计中使用的模块列表。它适合用作为地址筛选文件的起点。

示例

将二进制文件转换为无注解的源代码

```
cdoutil -output-file test.txt test.bin
```

输出示例：

```
version 2.0
write 0xfc50000 0
write 0xfc50010 0
write 0xfc50018 0x1
write 0xfc5001c 0
write 0xfc50020 0
write 0xfc50024 0xffffffff
```

将二进制文件转换为含注解的源代码

```
cdoutil -annotate -output-file test.txt test.bin
```

输出示例：

```
version 2.0
# PCIEA_ATTRIB_0.MISC_CTRL.slverr_enable[0]=0x0
write 0xfcfa50000 0
# PCIEA_ATTRIB_0.ISR.{dpll_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x0}
write 0xfcfa50010 0
# PCIEA_ATTRIB_0.IER.{dpll_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x1}
write 0xfcfa50018 0x1
# PCIEA_ATTRIB_0.IDR.{dpll_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x0}
write 0xfcfa5001c 0
# PCIEA_ATTRIB_0.ECO_0.eco_0[31:0]=0x0
write 0xfcfa50020 0
# PCIEA_ATTRIB_0.ECO_1.eco_1[31:0]=0xffffffff
write 0xfcfa50024 0xffffffff
```

编辑二进制 CDO 文件

```
cdoutil -annotate -output-file test.txt test.bin
vim test.txt
cdoutil -output-binary-be -output-file test-new.bin test.txt
```

确保 .bif 文件当前使用的是 test-new.bin 而不是 test.bin，然后重新运行 Bootgen 以创建 .pdi 文件。

将源代码转换为二进制文件

```
cdoutil -output-binary-be -output-file test.bin test.txt
```

Bootgen 设计咨询

- AMD 建议您为现场系统生成您自己的密钥，然后将这些密钥提供给开发者的工具。请参阅[答复记录 76171](#) 了解更多信息。
- 在此版本中，Versal 自适应 SoC 支持几个加密密钥滚动块。请参阅[答复记录 76515](#) 了解更多信息。
- 从 Versal 自适应 SoC 2022.2 起：为了减小 PLM 并确保使其大小适合 PPU RAM，允许的最大分区数从 32 个减少到 20 个，允许的最大镜像/子系统数从 32 个减少到 10 个。如超出此限制，Bootgen 会在创建启动镜像时报错退出。

禁用此错误的选项是使用 `BOOTGEN_SKIP_MAX_PARTITIONS_CHECK` 环境变量。

请确保处理 PLM 代码中的更改，然后继续创建含任意数量的分区/镜像的 PDI。

- 不支持串联/部分比特流处理：请参阅[答复记录 35054](#) 获取更多详细信息。
- AMD Versal™ Bootgen 支持更新：请参阅[答复记录 34634](#) 获取更多详细信息。
- 在 bif 中使用启动镜像时，您可能会看到以下警告：

```
[WARNING]: ID code is taken from base PDI, ignoring 'id_code' specified  
in the BIF
```

与此警告相反，Bootgen 使用的是来自 BIF 的 `id_code`，而不是 `base pdi`。AMD 建议在 bif 中使用适当的 `id_code`。

附加资源与法律声明

查找其他文档

技术信息门户网站

AMD 技术信息门户网站是旨在使用您的网页浏览器提供健全的文档搜索和导航的在线工具。要访问该技术信息门户网站，请转至 <https://docs.amd.com>。

注释：单击链接将打开英语版本，但您可从下拉列表中选择简体中文版本（如可用）。请注意，简体中文版本可能比英语版本旧。

Documentation Navigator

Documentation Navigator (DocNav) 是预安装的工具，支持访问 AMD 自适应计算文档、视频和支持资源，您可在其中通过筛选和搜索来查找信息。要打开 DocNav，请执行以下操作：

- 在 AMD Vivado™ IDE 中，单击“Help” → “Documentation and Tutorials”。
- 在 Windows 上，单击“Start”（开始）按钮并选中“Xilinx Design Tools” → “DocNav”。
- 在 Linux 命令提示中输入 docnav。

注释：如需了解有关 DocNav 的更多信息，请参阅《Documentation Navigator 用户指南》([UG968](#))。

注释：您无法从 DocNav 访问简体中文版本。请使用设计中心网页。

设计中心

AMD 设计中心提供了根据设计任务和其他主题整理的文档链接，可供您用于了解关键概念以及常见问题解答。要访问设计中心，请执行以下操作：

- 在 DocNav 中，单击“Design Hubs View”选项卡。
- 转至[设计中心](#)网页。

支持资源

如需获取答复记录、技术文档、下载以及论坛等支持资源，请访问[技术支持](#)。

参考资料

以下文档提供了有关支持 Bootgen 进程的各项进程的附加信息：

1. 《Zynq 7000 SoC 技术参考手册》([UG585](#))
2. 《Zynq 7000 SoC 软件开发者指南》([UG821](#))
3. 《Vivado Design Suite 用户指南：版本说明、安装和许可》([UG973](#))
4. 《Zynq UltraScale+ MPSoC：软件开发指南》([UG1137](#))
5. 《Zynq UltraScale+ 器件技术参考手册》([UG1085](#))
6. 《Zynq 7000 SoC 安全启动入门指南》([UG1025](#))
7. 《Versal 自适应 SoC 系统软件开发者指南》([UG1304](#))
8. 《嵌入式设计教程：Versal 自适应计算加速平台》([UG1305](#))
9. 《Versal 自适应 SoC 技术参考手册》([AM011](#))
10. [AMD Zynq UltraScale+ MPSoC 解决方案中心](#)
11. 《Vitis 统一软件平台文档：嵌入式软件开发》([UG1400](#))
12. 《Zynq UltraScale+ MPSoC：嵌入式设计教程》([UG1209](#))
13. 《Zynq 7000 SoC 安全启动》([XAPP1175](#))
14. 《Zynq 7000 SoC 系统存储器的运行时间完整性和身份验证检查》([XAPP1225](#))
15. 《BBRAM 和 eFUSE 烧录》([XAPP1319](#))
16. 《Versal 自适应 SoC 安全手册》(UG1508)。本手册需要从[设计安全性专区](#)下载有效的 NDA。
17. 《Versal 自适应 SoC 寄存器参考资料》([AM012](#))

修订历史

下表列出了本文档的修订历史。

章节	修订综述
2024 年 12 月 13 日 2024.2 版	
BIF 语法和受支持的文件类型	更新注释。
HSM 模式步骤	移除一条注释。
boot_device	将 emmc/ mmc 更改为 mmc。
dump	更新描述。
对 Zynq 7000 器件分区进行加密	更新密钥生成的代码块。
属性	将 cmmc/ mmc 更改为 mmc。
2024 年 11 月 13 日 2024.2 版	
read	更新可选数据的内容
附录 E: Bootgen 设计咨询	添加内容。

章节	修订综述
dump	更新描述和示例。
optionaldata	更新描述。
boot_device	更新内容。
演示 emmc/mmc 作为含多个文件系统分区的辅助启动器件的 BIF 示例	新增章节
userkeys	添加注释
2024 年 5 月 30 日 2024.1 版	
附录 B: BIF 属性参考	添加以下相关信息 <ul style="list-style-type: none">· enable_auth_opt· optionaldata
2023 年 12 月 13 日 2023.2 版	
不适用	仅作编辑更改。
2023 年 10 月 18 日 2023.2 版	
第 2 章: 启动镜像布局	更新 GUI 图示和常规漏洞修复。
2023 年 7 月 26 日 2023.1 版	
常规更新	仅作编辑更新。无技术内容更新。
2023 年 6 月 15 日 2023.1 版	
常规更新	仅作编辑更新。无技术内容更新。
2023 年 5 月 16 日 2023.1 版	
imagedstore	添加主题。
2023 年 1 月 2 日 2022.2 版	
常规更新	仅作编辑更新。无技术内容更新。
2022 年 12 月 23 日 2022.2 版	
第 6 章: SSIT 支持	常规更新。
2022 年 12 月 14 日 2022.2 版	
第 6 章: SSIT 支持	更新 Versal HBM 相关内容。
2022 年 10 月 19 日 2022.2 版	
第 6 章: SSIT 支持	添加 SSI 技术身份验证流程。
不适用	漏洞修复。
2022 年 4 月 26 日 2022.1 版	
第 4 章: 使用 Bootgen GUI	更新镜像
第 6 章: SSIT 支持	更新 Versal Premium 相关内容
附录 B: BIF 属性参考	更新 BIF 属性
附录 C: 命令参考	更新命令参考信息

请阅读：重要法律声明

本文档所示信息仅做参考，其中可能包含不准确的技术信息、疏漏和印刷错误。受诸多原因影响，此处所含信息可能发生更改，也可能无法准确呈现，这些原因包括但不限于产品和路线图变更、组件和主板版本更改、新增模型和/或产品发布、不同制造商之间存在的产品差异、软件更改、BIOS 刷新、固件升级等。任何计算机系统均存在安全性漏洞风险，无法彻底阻止也无法缓解这类风险。AMD 没有任何义务来更新或者以任何其他方式纠正或修改这些信息。但 AMD 保留随时修改这些信息和更改文档内容的权利，AMD 没有任何义务将此类修改或更改通知任何人。此处信息

“按原样”提供。AMD 对于本文档内容不作任何陈述或保证，并且对于这些信息中可能出现的任何不准确、错误或疏漏问题不承担任何责任。对于有关任何暗含的非侵权、适销性及适合特定用途的保证，AMD 特此声明不承担任何责任。无论在任何情况下，对于任何人因使用此处包含的任何信息而形成的依赖或者引发的任何直接、间接、特殊或其他后果性损害，AMD 概不负责，即使 AMD 已明确获悉存在发生此类损害的可能性也是如此。

关于与汽车相关用途的免责声明

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

版权声明

© Copyright 2019 - 2024 AMD 公司，版权所有。AMD、AMD 箭头标识、UltraScale、UltraScale+、Versal、Vitis、Vivado、Zynq 及其组合均为 Advanced Micro Devices, Inc. 的商标。“PCI”、“PCIe”和“PCI Express”均为 PCI-SIG 拥有的商标，且经授权使用。“AMBA”、“AMBA Designer”、“Arm”、“ARM1176JZ-S”、

“CoreSight”、“Cortex”、“PrimeCell”、“Mali”和“MPCore”为 Arm Limited 在美国和/或其他国家或地区的商标。此出版物中所使用的其他产品名称仅用于标识目的，可能是其各自所属公司的商标。