

Vitis Unified Software Platform Documentation

Embedded Software Development

UG1400 (v2024.1) May 30, 2024

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



Table of Contents

Section I: Getting Started with Vitis.....	11
 Chapter 1: Navigating Content by Design Process.....	12
 Chapter 2: Vitis Software Platform Release Notes.....	13
What's New.....	13
Embedded GNU Toolchain Details.....	13
Changed Behavior.....	13
Known Issues.....	14
Unified IDE Features versus Classic IDE Features.....	14
 Chapter 3: Installation.....	16
Installation Requirements.....	16
Vitis Software Platform Installation.....	18
 Chapter 4: Getting Started with the Vitis Software Platform.....	28
Vitis Unified Software Platform Overview.....	28
Migrating from the Classic Vitis IDE to Vitis Unified IDE.....	33
Section II: Using the Vitis Unified IDE.....	36
 Chapter 5: Launching the Vitis Unified IDE.....	37
Vitis Unified IDE Launch Options.....	38
 Chapter 6: Vitis Unified IDE View and Feature.....	40
Vitis Component View.....	41
Search View.....	43
Source Control.....	45
Debug View.....	48
Example View.....	50
Code View and Smart Editor.....	52
Preferences.....	54

Parallel Compiling.....	57
Notification for File Change.....	57
New Feature Preview.....	57
Chapter 7: Develop.....	59
Managing Platforms and Platform Repositories.....	59
Target Platform.....	60
Applications.....	74
Using Custom Libraries in Application Projects.....	84
Chapter 8: Run, Debug, and Optimize.....	85
Launch Configurations.....	85
Target Connections.....	88
Running the Application Component.....	91
Debugging Application Component.....	92
Cross-Triggering.....	104
Profile/Analyze.....	115
Optimize: Performance Analysis.....	118
Creating a Boot Image.....	118
Programming Flash.....	121
Multi-Cable and Multi-Device Support.....	123
Chapter 9: User Managed Flow.....	125
Setting User Specified Tool Chain.....	128
Chapter 10: Vitis Utilities.....	129
Software Command-Line Tool.....	129
Program Device.....	129
Vitis Terminal.....	130
Project Export and Import.....	130
Generating Device Tree.....	132
Section III: Bootgen Tool.....	133
Chapter 11: Introduction.....	134
Installing Bootgen.....	134
Boot Time Security.....	135
Chapter 12: Boot Image Layout.....	136

Zynq 7000 SoC Boot and Configuration.....	136
Zynq UltraScale+ MPSoC Boot and Configuration.....	146
Versal Adaptive SoC Boot Image Format.....	159
Chapter 13: Creating Boot Images.....	175
Boot Image Format (BIF).....	175
BIF Syntax and Supported File Types.....	176
Attributes.....	181
Chapter 14: Using Bootgen GUI.....	191
Launch Bootgen GUI	191
Bootgen GUI for Zynq 7000 and Zynq UltraScale+ Devices.....	192
Using Bootgen GUI Options for Versal Adaptive SoCs.....	193
Using Bootgen on the Command Line.....	196
Commands and Descriptions.....	197
Chapter 15: Boot Time Security.....	201
Using Encryption.....	202
Using Authentication.....	214
Versal Authentication Support.....	225
Versal Hashing Scheme.....	227
Using HSM Mode.....	227
Chapter 16: SSIT Support.....	257
Chapter 17: FPGA Support.....	269
Encryption and Authentication.....	269
HSM Mode.....	270
HSM Flow with Both Authentication and Encryption.....	273
Chapter 18: Use Cases and Examples.....	275
Zynq MPSoC Use Cases.....	275
Versal Adaptive SoC Use Cases.....	285
Chapter 19: BIF Attribute Reference.....	296
aarch32_mode.....	296
aeskeyfile.....	297
alignment.....	300
auth_params.....	301

authentication.....	303
big_endian.....	305
bbram_kek_iv.....	306
bh_kek_iv.....	306
bh_keyfile.....	306
bh_key_iv.....	308
bhsignature.....	308
blocks.....	309
boot_config.....	311
boot_device.....	312
bootimage.....	314
bootloader.....	316
bootvectors.....	317
checksum.....	318
copy.....	319
core.....	319
delay_auth.....	320
delay_handoff.....	321
delay_load.....	321
destination_cpu.....	322
destination_device.....	323
early_handoff.....	323
efuse_kek_iv.....	324
efuse_user_kek0_iv.....	324
efuse_user_kek1_iv.....	325
encryption.....	325
exception_level.....	327
familykey.....	328
file.....	329
fsbl_config.....	329
headersignature.....	330
hivec.....	331
id.....	332
image.....	334
imagestore.....	334
init.....	335
keysrc.....	336
keysrc_encryption.....	337

load.....	338
metaheader.....	339
name.....	340
offset.....	341
optionaldata.....	342
overlay_cdo.....	342
parent_id.....	343
partition.....	343
partition_owner, owner.....	345
pid.....	346
pmufw_image.....	346
ppkfile.....	347
presign.....	348
pskfile.....	349
puf_file.....	350
reserve.....	351
split.....	352
spkfile.....	353
spksignature.....	354
spk_select.....	355
sskfile.....	356
startup.....	357
trustzone.....	358
type.....	359
udf_bh.....	360
udf_data.....	361
userkeys.....	361
xip_mode.....	364
Chapter 20: Command Reference.....	365
arch.....	365
authenticatedjtag.....	366
bif_help.....	366
dual_ospo_mode.....	366
dual_qspi_mode.....	367
dump.....	368
dump_dir.....	368
efuseppkbits.....	369

enable_auth_opt.....	369
encrypt.....	369
encryption_dump.....	370
fill.....	370
generate_hashes.....	371
generate_keys.....	372
h, help.....	373
image.....	373
log.....	374
nonbooting.....	375
o.....	375
p.....	376
padimageheader.....	376
process_bitstream.....	377
read.....	377
spksignature.....	378
split.....	378
verify.....	379
verify_kdf.....	379
w.....	380
zynqmpes1.....	380
Initialization Pairs and INT File Attribute.....	381
Chapter 21: CDO Utility.....	383
Accessing.....	383
Usage.....	383
Examples.....	384
Chapter 22: Design Advisories for Bootgen.....	386
Section IV: Vitis Python CLI.....	387
Chapter 23: Python Vitis Commands.....	394
Python API: A command-line tool for creating and managing projects in Vitis.....	388
Managing Vitis IDE Components through Python APIs.....	389
System Project.....	390
Script Building Logger: Tool to Automate Script Creation Based on IDE Actions.....	391
Chapter 24: Python XSDB Commands.....	395

Chapter 25: Python XSDB Usage Examples.....	396
Section V: Software Command-Line Tool.....	409
Chapter 26: Software Command-Line Tool.....	410
Chapter 27: XSCT Commands.....	412
Target Connection Management.....	412
Target Registers.....	416
Program Execution.....	417
Target Memory.....	431
Target Download FPGA/BINARY.....	438
Target Reset.....	441
IPI commands to Versal PMC.....	442
Target Breakpoints/Watchpoints.....	445
Jtag UART.....	451
Miscellaneous.....	453
JTAG Access.....	461
Target File System.....	470
SVF Operations.....	477
Device Configuration System.....	482
STAPL Operations.....	484
Vitis Projects.....	488
Chapter 28: XSCT Use Cases.....	541
Common Use Cases.....	541
Changing Compiler Options of an Application Project.....	542
Creating an Application Project Using an Application Template (Zynq UltraScale+ MPSoC FSBL).....	542
Creating an FSBL Application Project Using Manually Created Domain (Zynq UltraScale+ MPSoC FSBL).....	543
Creating a Bootable Image and Program the Flash.....	543
Debugging a Program Already Running on the Target.....	544
Debugging Applications on Zynq UltraScale+ MPSoC.....	545
Selecting Target Based on Target Properties.....	548
Memory and Register accesses from XSCT.....	548
Modifying BSP Settings.....	552
Performing Standalone Application Debug.....	552

Generating SVF Files.....	555
Program U-BOOT over JTAG.....	556
Running an Application in Non-Interactive Mode.....	556
Running Tcl Scripts.....	557
Switching Between XSCT and Vitis Integrated Design Environment.....	558
Using JTAG UART.....	558
Working with Libraries.....	559
Editing FSBL/PMUFW Source File.....	560
Editing FSBL/PMUFW Settings.....	560
Exchanging Files between Host Machine and Linux Running on QEMU.....	561
Loading U-Boot over JTAG.....	561
Chapter 29: Hardware Software Interface (HSI) Commands.....	564
XSCT Interface Examples.....	564
Microprocessor Software Specification (MSS).....	578
Microprocessor Library Definition (MLD).....	585
Microprocessor Driver Definition (MDD).....	597
Microprocessor Application Definition (MAD).....	609
HSI Commands.....	612
Section VI: GNU Compiler Tools.....	646
Chapter 30: Overview.....	647
Chapter 31: Compiler Framework.....	648
Chapter 32: Common Compiler Usage and Options.....	650
Usage.....	650
Input Files.....	650
Output Files.....	651
File Types and Extensions.....	651
Libraries.....	652
Language Dialect.....	652
Commonly Used Compiler Options: Quick Reference.....	653
General Options.....	654
Library Search Options.....	656
Header File Search Option.....	656
Default Search Paths.....	656
Linker Options.....	657

Memory Layout.....	658
Object-File Sections.....	659
Linker Scripts.....	662
Chapter 33: MicroBlaze Compiler Usage and Options.....	664
MicroBlaze Compiler.....	664
Processor Feature Selection Options.....	664
General Program Options.....	667
MicroBlaze Application Binary Interface.....	669
MicroBlaze Assembler.....	669
MicroBlaze Linker Options.....	670
MicroBlaze Linker Script Sections.....	671
Tips for Writing or Customizing Linker Scripts.....	671
Startup Files.....	672
Modifying Startup Files.....	675
Compiler Libraries.....	677
Thread Safety.....	678
Command Line Arguments.....	678
Interrupt Handlers.....	678
Chapter 34: Arm Compiler Usage and Options.....	680
Usage.....	680
Chapter 35: Other Notes.....	682
C++ Code Size.....	682
C++ Standard Library.....	682
Position Independent Code (Relocatable Code).....	683
Other Switches and Features.....	683
Section VII: Embedded Design Tutorials.....	684
Section VIII: Drivers and Libraries.....	685
Appendix A: Additional Resources and Legal Notices.....	686
Finding Additional Documentation.....	686
Support Resources.....	687
Revision History.....	687
Please Read: Important Legal Notices.....	691

Getting Started with Vitis

This section provides a brief overview of the AMD Vitis™ unified software platform and describes the installation requirements and procedures to install and run the tool.

This section contains the following chapters:

- [Navigating Content by Design Process](#)
- [Vitis Software Platform Release Notes](#)
- [Installation](#)
- [Getting Started with the Vitis Software Platform](#)

Navigating Content by Design Process

AMD Adaptive Computing documentation is organized around a set of standard design processes to help you find relevant content for your current development task. You can access the AMD Versal™ adaptive SoC design processes on the [Design Hubs](#) page. You can also use the [Design Flow Assistant](#) to better understand the design flows and find content that is specific to your intended design needs.

- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
 - [Creating a Platform Component from XSA](#)
 - [Customizing a Pre-Built Platform](#)
 - [Creating an Application Component](#)
 - [Run, Debug, and Optimize](#)

For other design processes, refer to *Vitis Unified Software Platform Documentation Landing Page* ([UG1416](#))

Vitis Software Platform Release Notes

What's New

For information about what's new in this version of the AMD Vitis™ unified software development platform, see the [Vitis What's New Page](#).

Embedded GNU Toolchain Details

The following GNU toolchain components are installed with the Vitis unified software platform:

- **binutils:** 2.39
 - **gcc:** 12.2
 - **gdb:** 12.1
 - **glibc:** 2.36
 - **newlib:** 4.2
-

Changed Behavior

The following table specifies differences between this release and prior releases that impact behavior or flow when migrating.

Table 1: Changed Behavior Summary

Area	Behavior
Vitis IDE	Parallel Build is changed to be off by default and can be manually turned on in Preferences.
	The Acceleration Examples in the Example view is renamed to System Design Examples. The examples in the Vitis Accelerated Libraries Repository is spread into multiple top-level categories including AI Engine Examples, HLS Examples, and System Design Examples.
	Vitis IDE now calls <code>v++ -mode hls</code> instead of <code>vitis_hls</code> for compiling HLS designs.
Vitis HLS	The <code>vitis_hls</code> command now launches Vitis for HLS (if Vitis is installed). To keep on using the deprecated user interface of HLS, invoke <code>vitis_hls -classic</code> .
	Similar to 2024.1, several Vitis HLS features require a free license (For example, Code Analyzer).

Known Issues

Known issues for the Vitis software platform are available on the [Vitis 2024 Known Issues](#) web page.

Unified IDE Features versus Classic IDE Features

Following table showcases the support status in both Unified IDE and Classic IDE.

Table 2: Example table

Features	Vitis Unified IDE	Vitis Classic IDE
Platform Creation	Target Platform	Target Platform
Application Development	Applications	Applications
Using Customer Libraries	No Support	Using Customer Libraries in Application
Run Application	Running the Application Component	Run Application Project
Debug Application	Debugging Application Component	Debug Application Project
Cross Trigger	Cross-Triggering	Cross-Triggering
TCF Profiling	TCF Profiling	TCF Profiling
Gprof Profiling	No Support	gprof Profiling
OS Aware Debug	No support	OS Aware Debug
Xen Aware Debug	No support	Xen Aware Debug
OPTimize: Performance Analysis	No Support	Optimize Performance Analysis
Creating Boot Image	Creating a Boot Image	Creating a Boot Image
Program Flash	Programming Flash	Programming Flash
Multi-Cable and Multi-Device Support	Multi-Cable and Multi-Device Support	Multi-Cable and Multi-Device Support

Table 2: Example table (cont'd)

Features	Vitis Unified IDE	Vitis Classic IDE
Target Management	Target Connections	Target Connection
Version Control	Source Control	Version Control with Git
User Managed Flow	User Managed Flow	No Support
Workspace Import and Export	Project Export and Import	Project Export and Import
Software Command Tool	Section V: Software Command-Line Tool	Software Command Line Tool
Python CLI	Section IV: Vitis Python CLI	No Support

Installation

Installation Requirements

The AMD Vitis™ unified software platform consists of an integrated design environment (IDE) for interactive project development, and command-line tools for scripted or manual application development.

Installer Types

AMD provides two types of installers to support your installation requirements for the Vitis software platform.

- AMD Unified Installer for FPGAs & Adaptive SoCs
- AMD Vitis Embedded Installer

The AMD Unified Installer contains almost everything. It includes the full featured Vitis software platform, the AMD Vivado™ Design Suite and PetaLinux. You can choose which components to install during installation procedure.

The AMD Vitis Embedded Installer contains the tools for embedded development only. It contains the embedded software development version of Vitis Unified IDE and the utilities like XSCT and program_flash.

To develop an embedded software with Vitis software platform, you can download AMD Unified Installer for FPGAs & Adaptive SoCs or AMD Vitis Embedded Installer.

Vitis Unified Installer provides the following installation options:

- Run Vitis Embedded Installer only
- Install Vitis Embedded as an optional component during Vivado installation.

For each installer, AMD might provide different types of packages.

- The web installer only downloads essential components according to your installation requirements to speed up the download process.

- The Single File Download (SFD) provides a self-contained package for AMD components. Network downloading is not required during the installation process.

Note: You might still need network to upgrade system library to fulfill the installation requirements.

Requirements and Setup

Supported Operating System

Table 3: Embedded Software Development Flow Minimum System Requirements

Component	Requirement
	Development (Build Machine OS)
Operating system	<p>Linux, 64-bit:</p> <ul style="list-style-type: none">• Red Hat Enterprise Linux 7.4-7.7, 7.9: 64-bit• Red Hat Enterprise Linux 8.5-8.8, 9.0-9.3: 64-bit (Not supported for PetaLinux)• CentOS Linux 7.4-7.7, 7.9: 64-bit• AlmaLinux 8.7, 9.1: 64-bit• Ubuntu Linux 20.04.4-20.04.6, 22.04-22.04.3 LTS: 64-bit. Additional library installation required.• Amazon Linux 2 AL2 LTS: 64-bit (Not supported for PetaLinux)• SUSE Enterprise Linux 12.5, 15.3, 15.4: 64-bit (Not supported for PetaLinux) <p>Windows, 64-bit:</p> <ul style="list-style-type: none">• Windows 10: 22H2• Windows 11: 22H2
System memory	32 GB (64 GB is recommended)
Internet connection	Required for downloading drivers and utilities.
Hard disk space	<ul style="list-style-type: none">• Full Vitis installation: 200GB• Vitis Embedded Development: 20GB

Operating Systems End-of-Life Notification

Note: This release, 2024.1, is the last release that Vitis Software Development supports the following operating systems:

- Red Hat Enterprise Linux Workstation/Server/CentOS 7.4-7.7
- Red Hat Enterprise Linux 8.5-8.7
- Red Hat Enterprise Linux 9.0-9.1

- Ubuntu 20.04.4 LTS
- Ubuntu 22.04 LTS
- Windows 11.0 22H2

Note:

1. Ubuntu 22.04 LTS means the initial release of Ubuntu 22.04 LTS. The latest minor releases like Ubuntu 22.04.1 are still supported.
2. The latest versions of related major release (for example, RHEL 8, Ubuntu 20.04 LTS) are still supported.
3. Red Hat Enterprise Linux Workstation/Server/CentOS 7.9 will not be supported in 2025.1.

Required Libraries

You might need to install dependent libraries for certain Linux operating systems. The installation process might fail in case of missing libraries. In case of an error during the installation process, check xinstall.log file for more information.

Download the Installation File

For more information, refer to [Download Verification](#).

Vitis Software Platform Installation

Installing the Vitis Software Platform

Ensure your system meets all requirements described in [Installation Requirements](#).



TIP: To reduce installation time, disable anti-virus software and close all open programs that are not needed.

1. Go to the [AMD Adaptive Computing Downloads Website](#).
2. Download the installer for your operating system.
3. Run the installer, xsetup, or xsetup.exe, which opens the Welcome page. Extract the installer package.
4. Click **Next** to open the Select Install Type page of the Installer.
5. If installing with the web installer, enter your AMD user account credentials, and select **Download and Install Now** (only needed by web installer).
6. Click **Next** to open the Accept License Agreements page of the Installer.

7. Accept the terms and conditions by clicking each **I Agree** check box.
8. Click **Next** to open the Select Product to Install page of the Installer.
9. Select **Vitis** and click **Next** to open the Vitis Unified Software Platform page of the Installer.
10. Customize your installation by selecting design tools and devices (optional).

The default Design Tools selections are for standard Vitis Unified Software Platform installations, and include Vitis, Vivado, and Vitis HLS. You do not need to separately install Vivado tools. You can also install Model Composer and System Generator if needed.

You can enable **Vitis IP Cache** to install cache files for example designs found in the release. This is not required, but when selected, the files are installed at `<installdir>/Vitis/<release>/data/cache/xilinx`.

The default Devices selections are for devices used on standard acceleration platforms supported by the Vitis tools. You can disable some devices that might not be of interest in your installation.

11. Click **Next** to open the Accept License Agreements page of the Installer and accept as appropriate.
12. Click **Next** to open the Select Destination Directory page of the Installer.
13. Specify the installation directory, review the location summary, review the disk space required to insure there is enough space, and click **Next** to open the Installation Summary page of the Installer.
14. Click **Install** to begin the installation of the software.

After a successful installation of the full Vitis unified software, a confirmation message is displayed, with a prompt to run the `installLibs.sh` script.

1. Locate the script at: `<install_dir>/Vitis/<release>/scripts/installLibs.sh`, where `<install_dir>` is the location of your installation, and `<release>` is the installation version.

Note: This script is not required on Windows.

2. Run the script using `sudo` privileges as follows:

```
sudo installLibs.sh
```

The command installs a number of necessary packages for the Vitis tools based on the OS of your system.



IMPORTANT! Pay attention to any messages returned by the script. You might need to install any missing packages manually. For example, if your installation of Linux does not include the `zip` command-line utility, you need to manually install it. The utility is required by some of the Vitis tools and the `installLibs.sh` script does not install it for you.

Lightweight Installer Download

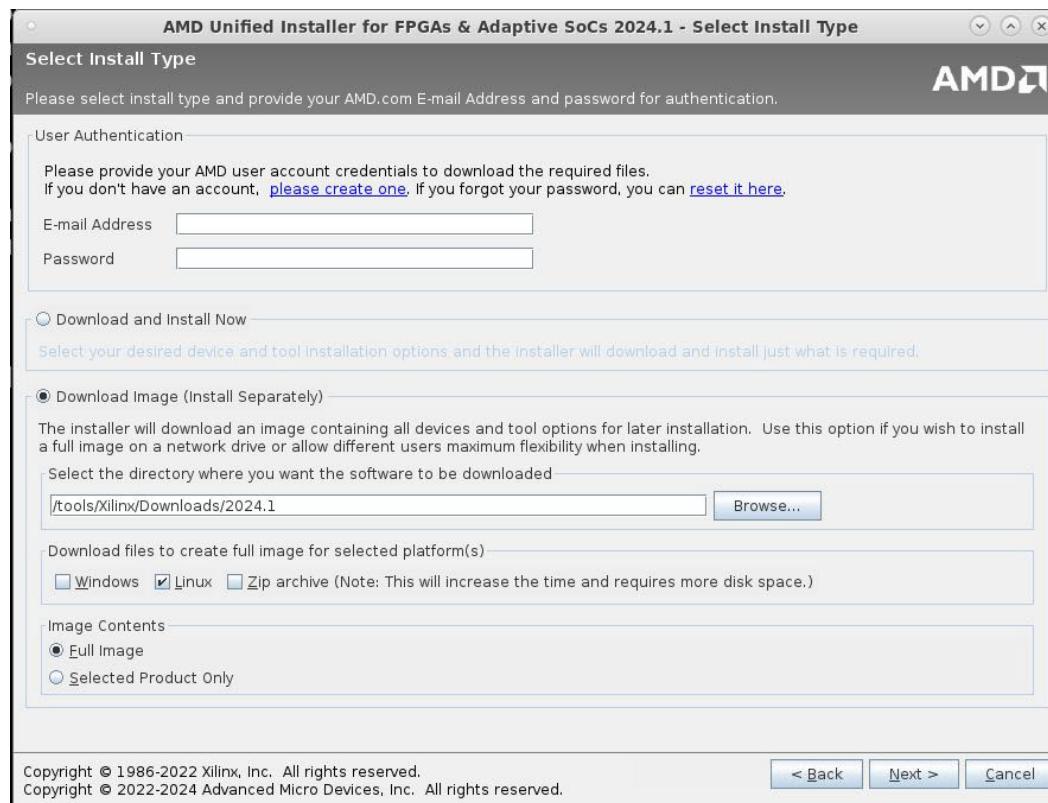
If you downloaded the lightweight installer, launch the downloaded file. You are prompted to log in and use your regular AMD login credentials to continue with the installation process.

Note: On Linux the file is a .bin file and can be launched by running `./<name of the file>.bin`. Ensure that you have changed the file permissions to execute.

After entering your login credentials, you can select between a traditional web-based installation or a full install image download.

- **Download and Install Now:** Allows you to select specific tools and device families on following screens, downloads only the files required to install those selections, and installs them for you.
- **Download Image (Install Separately):** Requires you to select a download destination and to choose whether you want a Windows only, Linux only, or an install that supports both operating systems. You also have an option to either download the full installer or download only selected products. There are no further options to choose with the Full Image selection, and installation needs to be done separately by running the xsetup application from the download directory.

Figure 1: AMD Unified Installer for FPGAs & Adaptive SoCs - Select Install Type



Prepare to Install the Tool

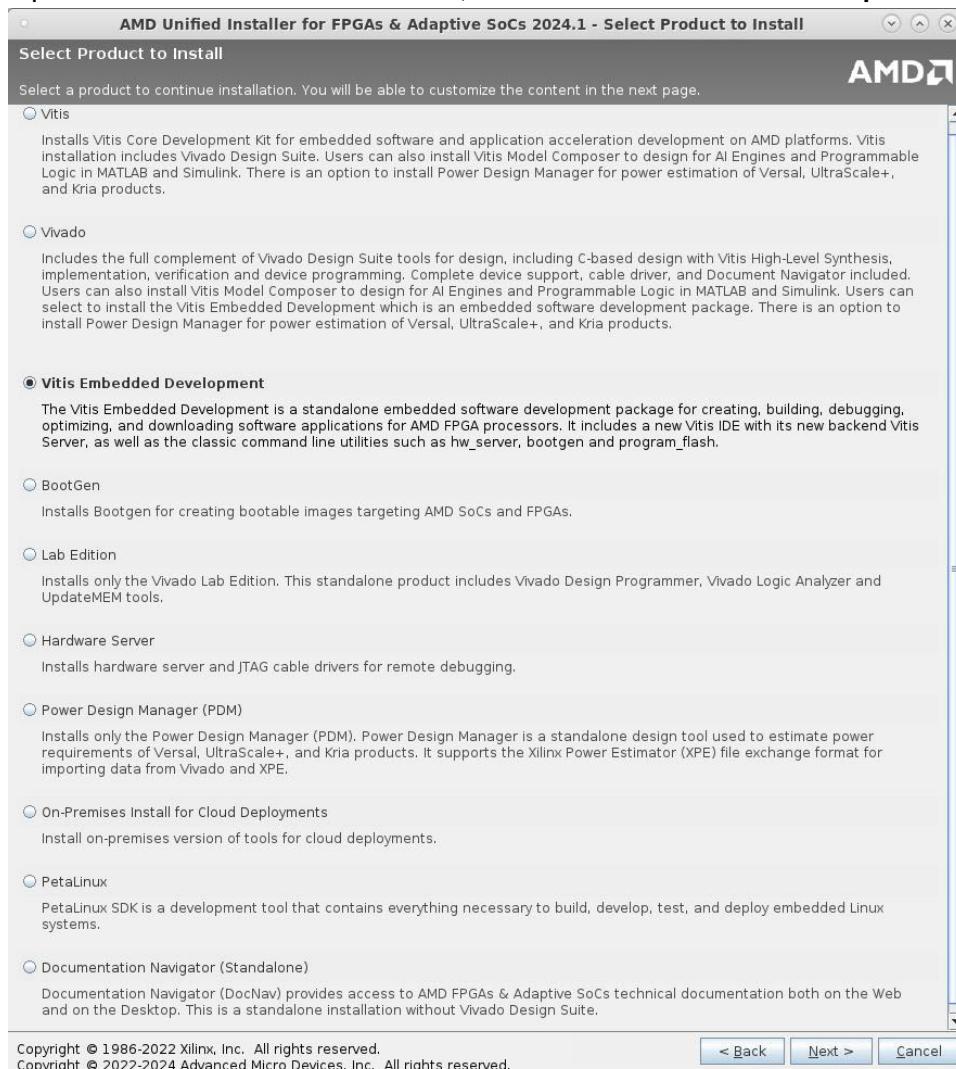
For more information on this, refer to [Prepare to Install the Tool](#).

Run the Installation File

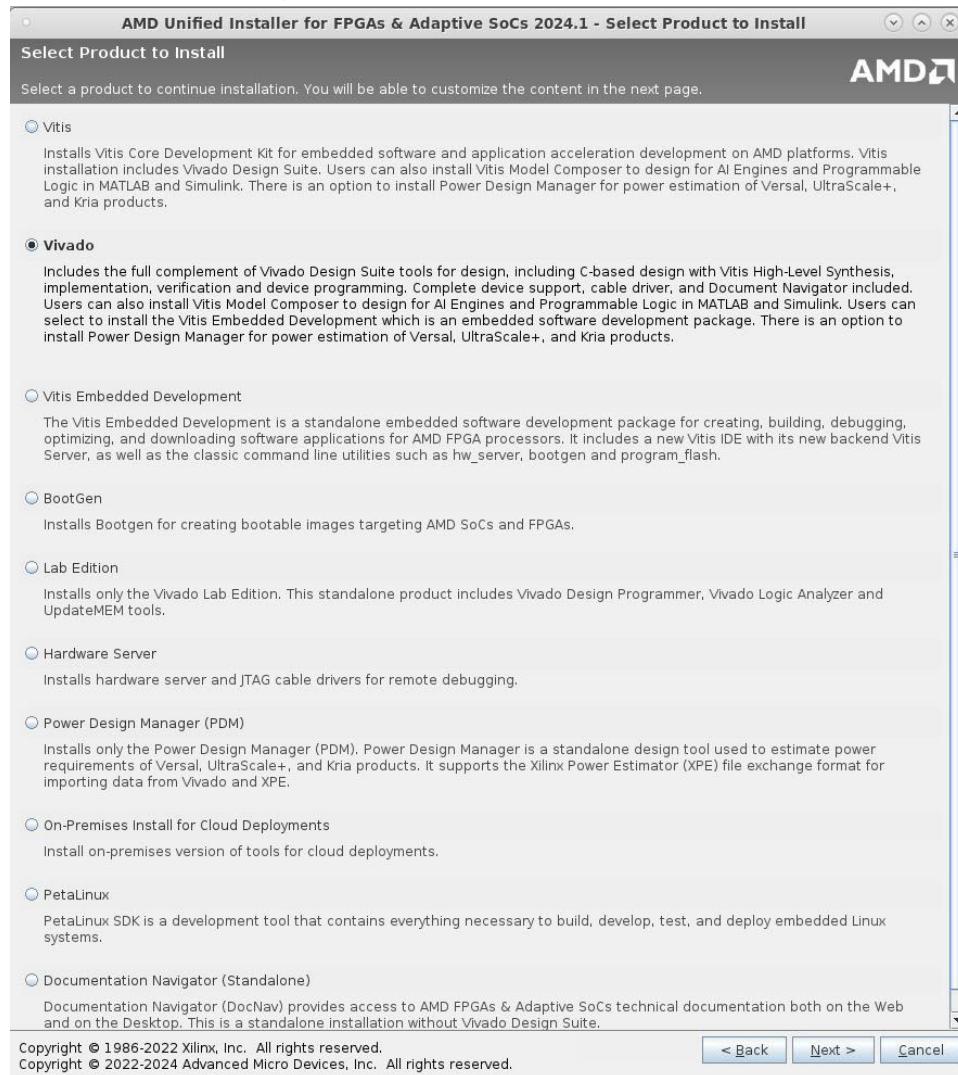
1. Run the installer, `xsetup` for Linux, or `xsetup.exe` for Windows, which opens the Welcome page.
2. Click **Next** to open the Select Install Type page of the Installer.
3. If installing with the web installer, enter your AMD user account credentials, and select **Download and Install Now** (only needed by web installer).
4. Click **Next** to open the Accept License Agreements page of the Installer.
5. Accept the terms and conditions by clicking each I Agree check box.
6. Click **Next** to open the Select Product to Install page of the Installer.

7. Select product to install

a. Option 1: To install Vitis Embedded, select **Vitis Embedded Development**.

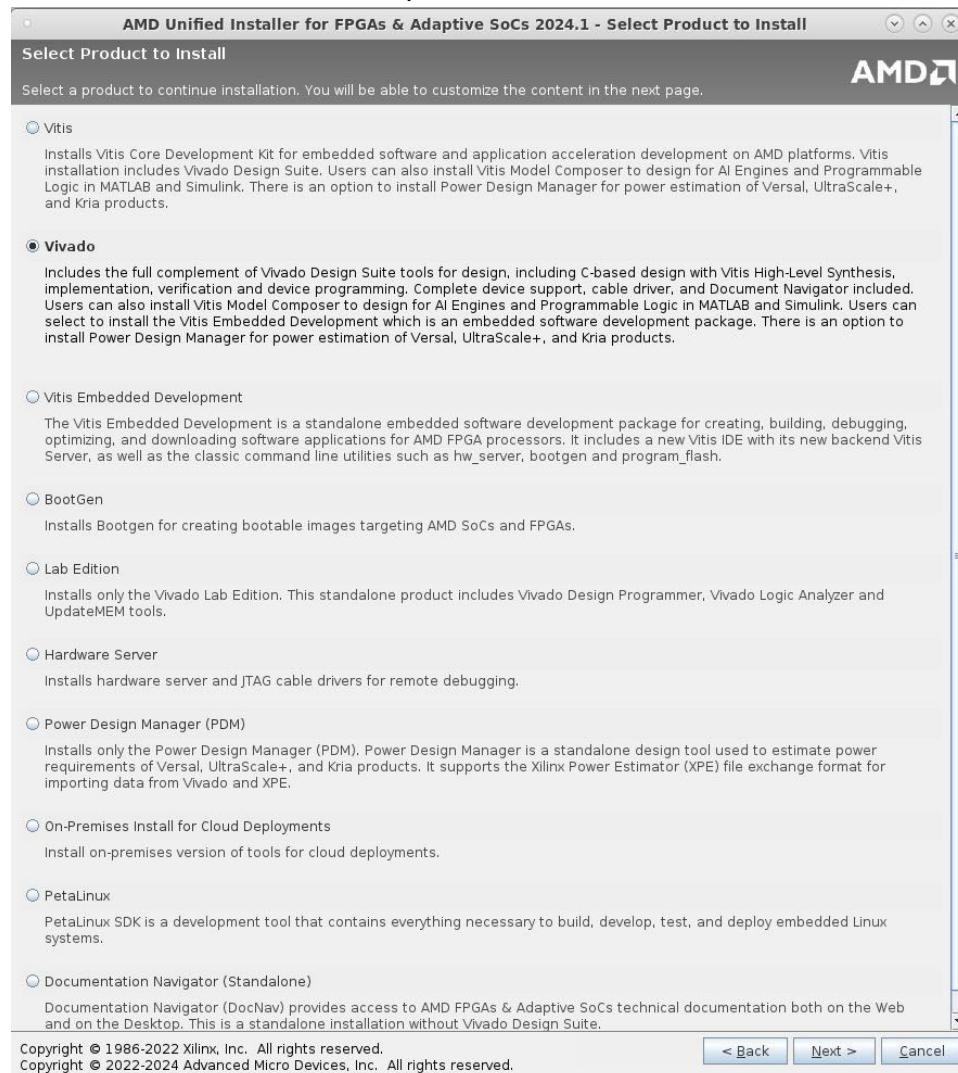


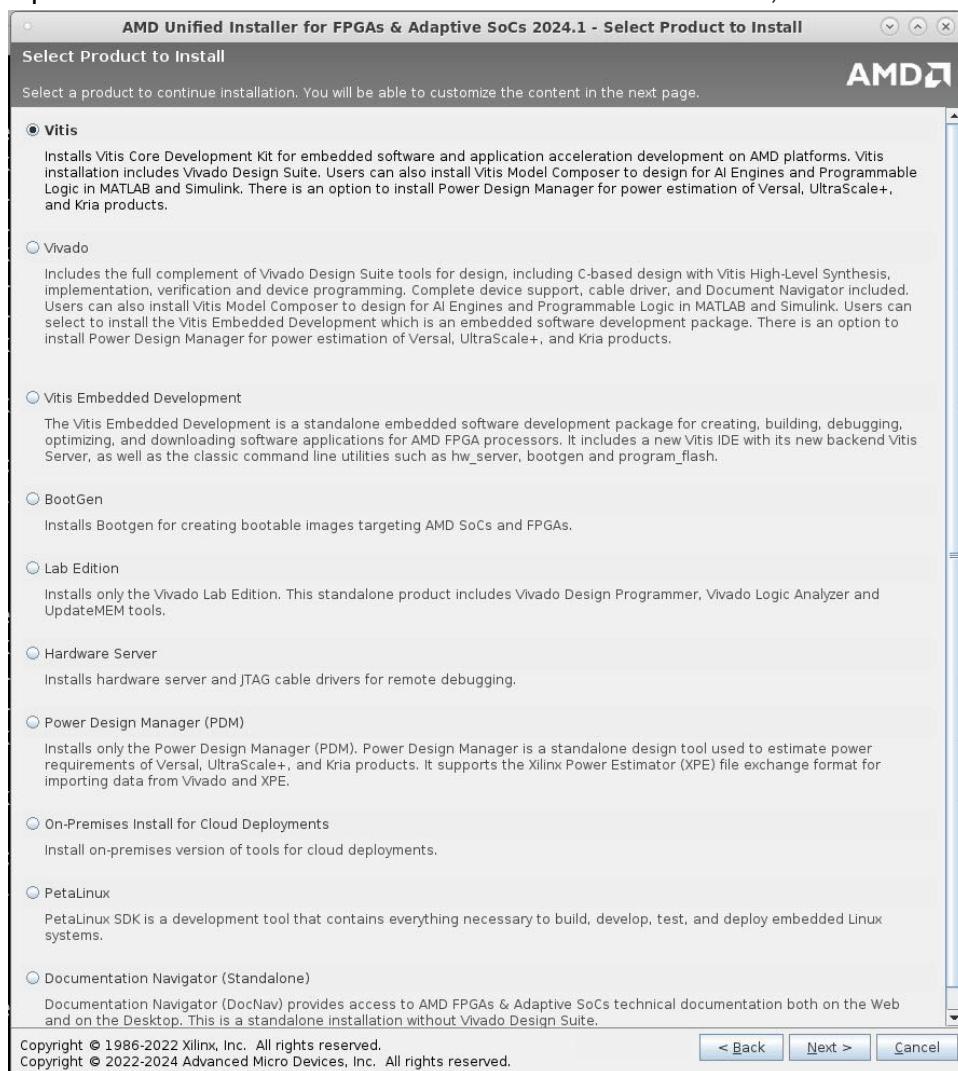
- b. Option 2: To install Vivado and Vitis Embedded, select **Vivado**, and enable the Vitis Embedded Installer option.



- i. Select Vivado ML Standard or Vivado ML Enterprise according to your license type.

ii. Enable Vitis Embedded Development.



c. Option 3: To install the full featured Vitis Software Platform, select **Vitis**.

8. Click **Next**.
9. Click **Next** to open the Accept License Agreements page of the installer and accept as appropriate.
10. Click **Next** to open the Select Destination Directory page of Installer.
11. Specify the installation directory, review the location summary, review the disk space required to ensure there is enough space, and click **Next** to open the **Installation Summary** page of the Installer.
12. Click **Install** to begin the installation of the software.

After a successful installation of the Vitis Software, a confirmation message is displayed, with a prompt to run the `installLibs.sh` script.

1. Locate the script at: <install_dir>Vitis/<release>/scripts/installLibs.sh, where <install_dir> is the location of your installation, and <release> is the installation version.

Note: This script is not required on Windows.

2. Run the script using `sudo` privileges as follows:

```
sudo ./installLibs.sh
```

The command installs a number of necessary package for the Vitis tools based on the OS of your system.



IMPORTANT! Pay attention to any messages returned by the script. You might need to install any missing packages manually. For example, if installation of Linux does not include the zip command-line utility, you need to manually install it. The utility is required by some of the Vitis tools and the `installLibs.sh` script does not install it for you.

Note: For more information about the AMD Unified Installer, refer to *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#)).

Setting Up the Environment to Run the Vitis Software Platform

Linux

To configure the environment to run the Vitis software platform, run the following script in a command shell to set up the tools to run in that shell:

```
#setup XILINX_VITIS and XILINX_VIVADO variables
source <Vitis_install_path>/Vitis/2024.1/settings64.sh
```



TIP: `.csh` scripts are also provided.

This sets up the tools for the Vitis embedded software development flow.

To run Vitis, type **vitis** and press enter in the Console.

If you have created a shortcut on your desktop, you can also double the shortcut to launch Vitis unified IDE.

Windows

To launch the Vitis software platform from Windows, do one of the following:

- Launch from a desktop button or Start menu command.

- From a Windows command shell, use `settings64.bat`:

```
C:> <VITIS_INSTALL_DIR>\VITIS\2024.1\settings64.bat
```

And launch: `vitis`.

Getting Started with the Vitis Software Platform

Vitis Unified Software Platform Overview

The AMD Vitis™ unified software platform combines all aspects of AMD software development into one unified environment. The Vitis software platform supports both the Vitis embedded software development flow and the Vitis application acceleration development flow, for software developers looking to use the latest in AMD FPGA-based software acceleration. This document discusses the embedded software development flow and use of Vitis core development kit.

The Vitis unified software platform contains many tools and utilities to support embedded software development as backend. It provides the Vitis unified integrated design environment (IDE) as the front end GUI to support embedded software developers work efficiently when developing software applications towards AMD embedded processors. The Vitis unified IDE works with hardware designs created with AMD Vivado™ Design Suite.

The Vitis unified IDE is a GUI refresh against the classic Eclipse based Vitis IDE. It adopts latest technology from Eclipse Foundation and uses Eclipse Theia as its base framework. The new framework enables faster GUI response, rich open-source community driven plugins and flexible configuration.

The Vitis unified IDE provides the following features for embedded software development:

- Creating platforms from AMD Vivado™ generated hardware designs and generating BSP for software development
- Creating applications from example designs or empty template
- Configuring and building the platforms and applications
- Running, debugging, or profiling the applications on hardware
- Managing multiple local or remote hardware connections with the Target Connection Manager
- Rich device and processor support, from MicroBlaze™, Zynq 7000, Zynq AMD UltraScale+™ MPSoC to AMD Versal™
- Creating boot images

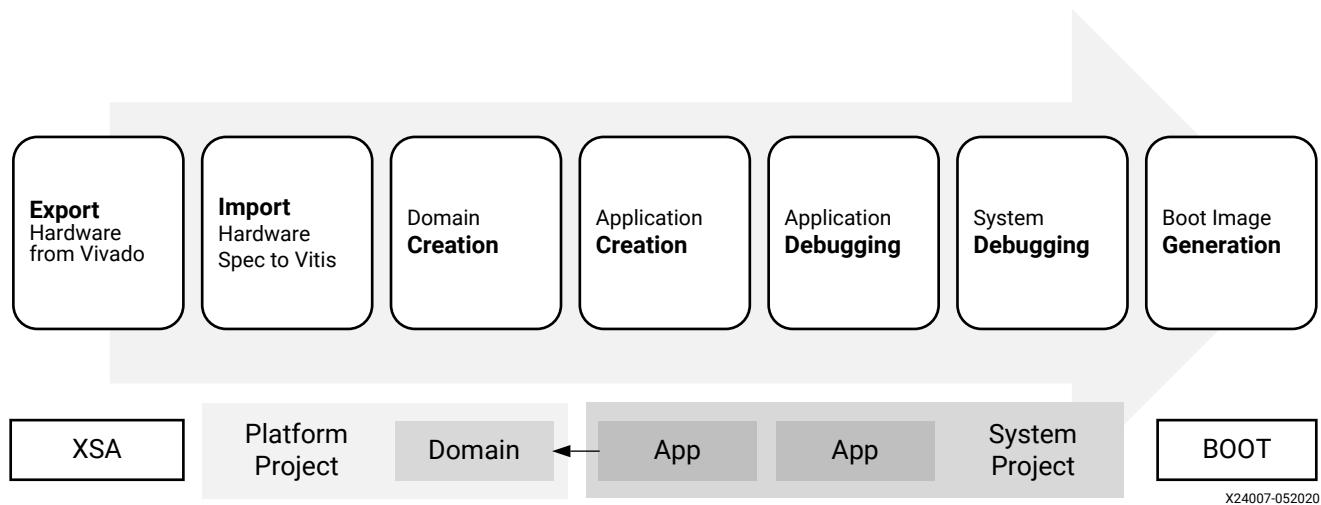
- Configuring devices
 - Programming Flash
 - Managing projects by IDE and run actions according to component type
 - Managing projects by user scripts and using IDE to assist debugging procedure
 - Source code version control with integrated git
 - All actions are supported in both GUI and command line interface (CLI)

The full Vitis installation includes both, the new IDE and classic IDE. You can launch the classic IDE with command `Vitis --classic`. A project migration utility is added to the classic Vitis IDE that allows you to migrate the classic Vitis IDE workspace to the new Vitis unified IDE.

Vitis Software Development Workflow

The following figure shows the embedded software application development workflow for the Vitis unified software platform.

Figure 2: Embedded Software Application Development Workflow



- Hardware engineers design the logic and export information required by software development from the AMD Vivado™ Design Suite to an XSA archive file.
 - Software developers import XSA into the Vitis software platform by creating a platform. Platform was heavily used by application acceleration projects. To unify the Vitis workspace architecture for all kinds of applications, software development projects now migrate to platform and application architecture. A platform includes hardware specification and software environment settings.
 - The software environment settings are called domains, which are also a part of a platform.
 - Software developers create applications based on the platform and domains.

- Applications can be debugged in the Vitis IDE.
- In a complex system, several applications run at the same time and communicate with each other. So the system level verification needs to be done as well.
- After everything is ready, the Vitis IDE can help to create boot images which initialize the system and launch applications.

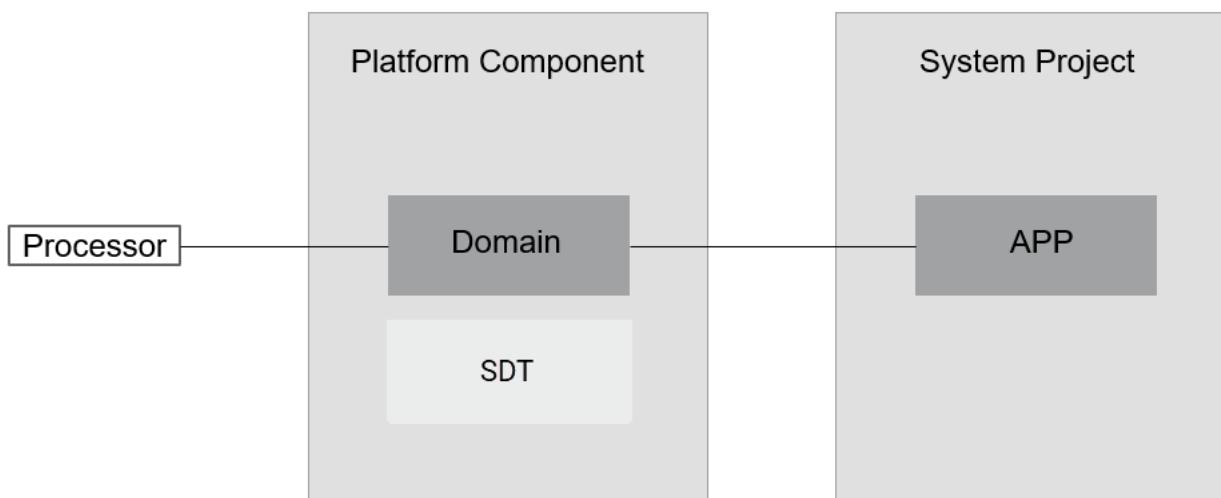
Note:

The Vitis Unified IDE offers support for two types of XSA and corresponding platforms: fixed XSA with a fixed platform and extensible XSA with an extensible platform. A fixed XSA, which includes a complete device image generated within Vivado, is employed to establish a fixed platform tailored for embedded development. On the other hand, the extensible XSA is used for acceleration development purposes. During the linking stage in V++, the comprehensive device image required for generating BOOT.bin for boot operations is created, specifically for acceleration development. So if you use extensible platform to create bare-metal application, you might encounter issue, for example "can not find corresponding devices." For more information about fixed platform and extensible platform, refer to [Fixed Platform vs Extensible Platform](#) in UG1393.

Workspace Structure in the Vitis Software Platform

There are two project types in Vitis unified workspace:

Figure 3: Vitis Software Platform Project Types



- **Workspace:** When you open the Vitis unified software platform, you create a workspace. A workspace is a directory location used by the Vitis unified software platform to store project data and metadata. An initial workspace location must be provided when the Vitis software platform is launched.

- **XSA:** XSA are exported from the Vivado Design Suite. It has the hardware specifications like processor configuration properties, peripheral connection information, address map, and device initialization code. You have to provide the XSA when creating a platform project.
- **SDT:** The System Device Tree is generated based on the XSA using the SDTGEN utility upon the Platform Creation. The SDT contains all processor and the respective memory mapped IP for each processor. The SDT also contains the top level memory. The SDT is not deployed on the target, it is purely used to capture and maintain the HW metadata from the XSA. The Lopper Utility is used to extract the HW metadata. This can include the processor list, the xparameters.h generation, linker script generated, and BSP creation, and so on.
- **Platform:** The target platform or platform is a combination of hardware components (XSA) and software components (domains/BSPs, boot components such as FSBL, and so on). Platforms in the repository are not editable. Platforms in the workspace are editable, and are referred to as platform components.
- **Platform Component:** A platform component is a project in Vitis Unified IDE to define a platform.
- **Domain:** A domain is a board support package (BSP) or the operating system (OS) with a collection of software drivers on which to build your application. The created software image contains only the portions of the AMD library you use in your embedded design. You can create multiple applications to run on the domain. A domain is tied to a single processor or a cluster of isomorphic processors (for example: A53_0 or A53) in the platform.
- **System Project :** A system project groups together applications that run simultaneously on a device. Two standalone applications for the same processor cannot sit together in a system project. Two Linux applications can sit together in a system project. A workspace can contain multiple system projects.
- **Application (Software Project):** A software project contains one or more source files, along with the necessary header files, to allow compilation and generation of a binary output (ELF) file. A system project can contain multiple application projects. Each software project must have a corresponding domain.

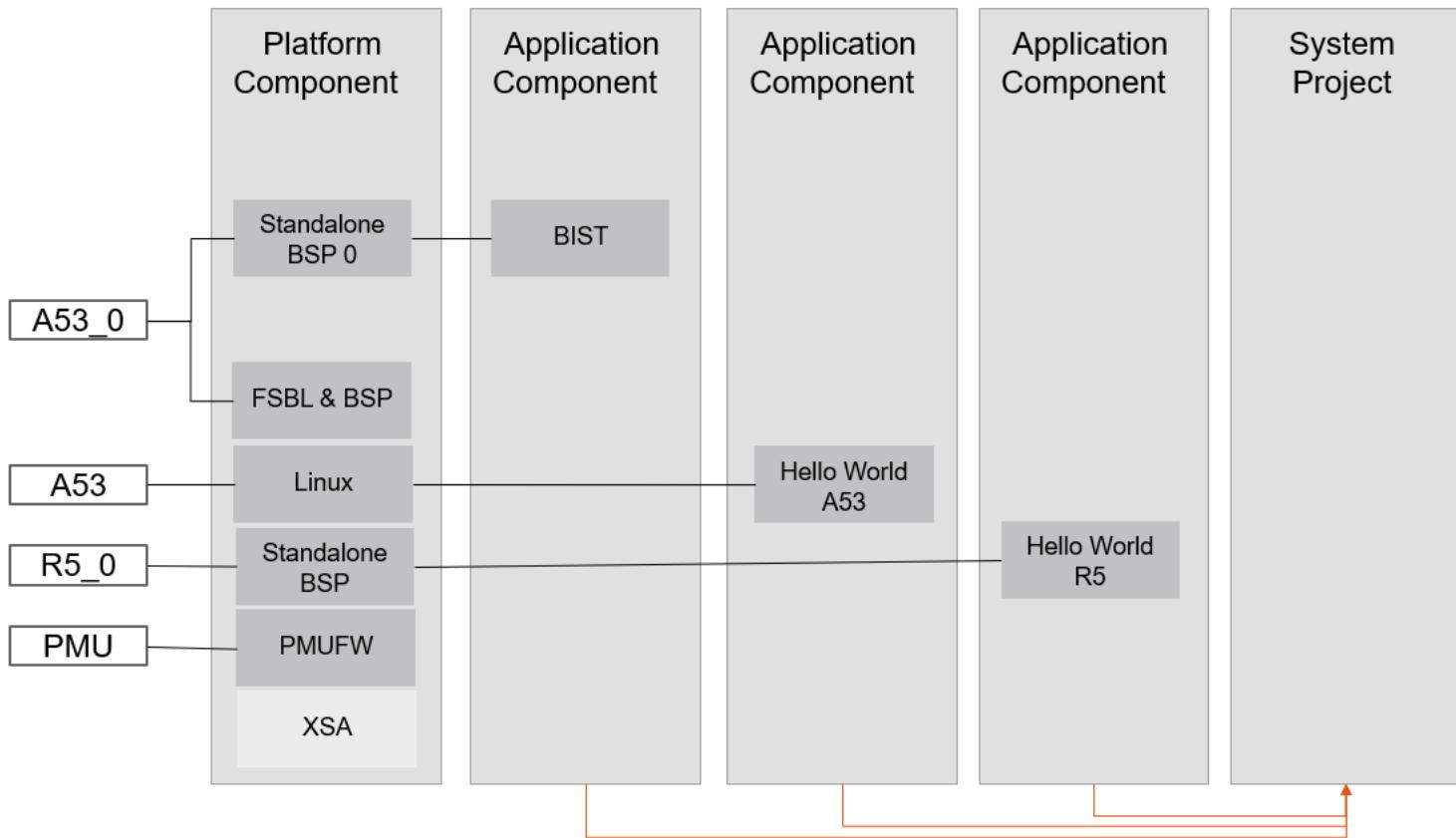
The Vitis platform has different configurations to support different use cases, outlined as follows:

- **Embedded:** Embedded platforms are fixed platforms. This platform only supports embedded software development for Arm® processors and MicroBlaze™ processors. The hardware design are not be modified by Vitis.
- **Embedded Acceleration:** Besides embedded software development, application acceleration is also supported on this type of platform. The platform provides clocks, bus interfaces, and interrupt controllers for the acceleration kernel to use.
- **Data Center Acceleration:** Acceleration kernels and x86 host applications can be developed on this platform. The kernel is controlled using a PCIe® bus.

Note: Vitis can extend the hardware design of extensible platforms, adding acceleration kernels (PL or AI Engine) to the original hardware design in the platform. It can also be used for software development.

The following is an example of a typical Vitis software development workspace for AMD Zynq™ UltraScale+™ MPSoC.

Figure 4: Vitis Software Development Workspace Example for Zynq UltraScale+ MPSoC



- Linux domains can be created for Arm® Cortex®-A53 SMP clusters. Linux applications can be compiled and linked against the `sysroot` of the Linux domain.
- Arm Cortex-A53 core 0 and Arm Cortex®-R5F core 0 can run hello world application at the same time, these two applications can be grouped into one system project.
- The bare metal build-in-self-test application on Arm Cortex-A53 core 0 can work in its own system project and have its own BSP settings.
- These system project is to manage multiple application components. Adding multiple application components in one system project means these applications would run at the same time. System project is not required if only one application runs at one time.
- Boot components such as FSBL and PMU firmware can be created in platform projects automatically. Boot components have their own BSP settings.

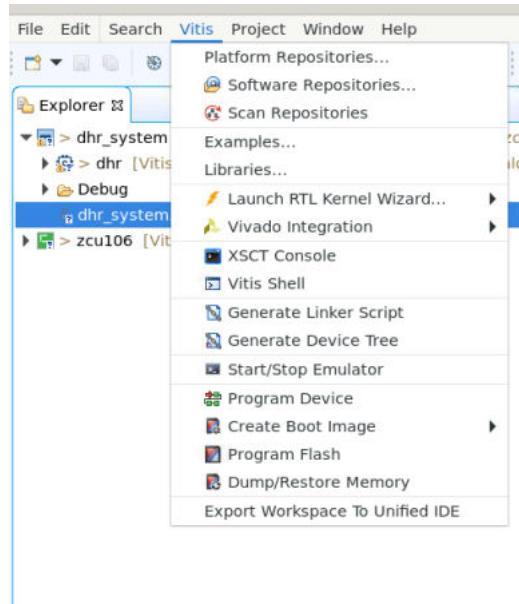
Migrating from the Classic Vitis IDE to Vitis Unified IDE

The Vitis Unified IDE workspace is designed to ensure version control optimization. Hence, the workspace and project metadata is incompatible with classic IDE. To facilitate the migration of projects, a new action is added to the classic IDE under **Vitis → Export Workspace to Unified IDE**.

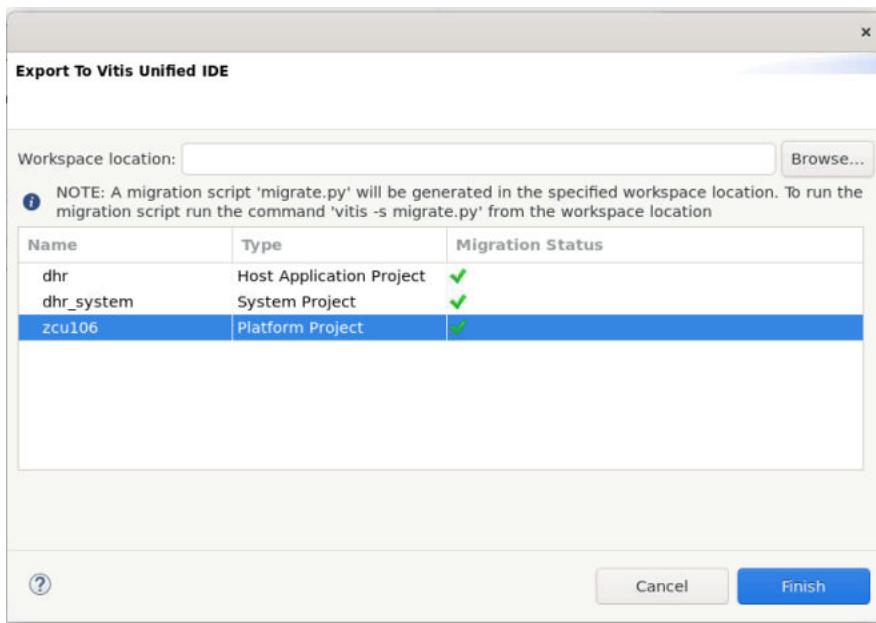
The tool launches a dialog with a list of all identified projects. All the projects have an associated status with them to inform you if the projects can be exported.

Execute the following steps to migrate a classic Vitis IDE project to Unified Vitis IDE.

1. Open the workspace with Classic Vitis IDE.
2. Select **Vitis → Export Workspace to Unified IDE**. Once you choose this, a dialog box appears.



3. In the dialog setup page, specify the migration target workspace location.



4. Click **Finish**. A migration script is generated in the specified location.
5. Open a terminal with Vitis environment settings, type the following command and hit enter.

```
vitis -s migrate.py
```

Vitis would run the migration script. The running process takes some time. Once completed, you can see the migrated components and projects in this workspace directory. All the project resources are copied to the new location.

Table 4: Supported Projects and Limitations

Project type	Limitation	Workaround
Platform	Local changes to BSP sources are not be migrated to the new workspace. A new BSP is created, and the settings are applied on the new BSP.	Copy the sources to the new BSP manually.
	Any embedded software repositories added to Vitis IDE cannot be migrated. A warning is shown in the wizard if you have any local sw repos.	All software repositories need to be migrated to lopper first. The path to the migrated repository can be added to the migration script (refer below screenshot for an example) to migrate the projects.
	IP drivers included in XSAs which were created with 2023.1 or older releases might encounter compilation errors.	Need to regenerate the XSA with 2024.1 release.

Table 4: Supported Projects and Limitations (cont'd)

Project type	Limitation	Workaround
Embedded Application	Applications referring to platforms that are outside the current workspace cannot be migrated.	Migrate the platform first and update the application to use the new platform before migrating the application.
	If your application relies on the device ID, some modifications might be required in the application source code. This is because the Device ID in the xparameter.h file has been deprecated. For more details, refer to <i>Porting Guide for embeddedsw Components System Device Tree Based Build Flow (UG1647)</i>	Refer to Standalone Application Component Migration Details
	Debug Configurations cannot be taken over to Unified IDE	Create launch configuration in Vitis Unified IDE

Note: If you wish to continue using the classic Vitis IDE, you can launch the classic Vitis IDE with the following command

```
vitis --classic
```

Note: To use classic Vitis IDE, install the full Vitis Software Platform.

For more information on Vitis Unified IDE, refer to [Launching the Vitis Unified IDE](#) and [Vitis Unified IDE View and Feature](#).

Standalone Application Component Migration Details

The base address of peripheral IP is used for all driver APIs instead of the DeviceID because using base address is an industry standard method to control the hardware. Because DeviceID is not generated in xparameters.h any more, compiling a migrated application directly might result in compilation errors. If your application relies on DeviceID for IP driver initialization, refer to AR: [Standalone Application Migration Details](#).

Using the Vitis Unified IDE

This section describes how to use the AMD Vitis™ Unified integrated design environment (IDE) to develop, run, debug, and optimize platforms and applications. The options in each view of the IDE are also explained. It also contains information about [Vitis Utilities](#).

This section contains the following chapters:

- [Vitis Unified IDE View and Feature](#)
- [Develop](#)
- [Run, Debug, and Optimize](#)
- [Vitis Utilities](#)

Launching the Vitis Unified IDE

This section explains the steps to launch the Vitis Unified IDE.

1. Use the following command to load the Vitis software platform environment.

```
source <Vitis_Installation_Directory>/settings64.sh
```

2. Use the following command to launch the Vitis Unified IDE.

```
vitis -w <workspace>
```

where `<workspace>` indicates a folder to hold all of the contents of your design project.

The workspace is used to group together the source and data files that make up a design, or multiple designs, and stores the configuration of the tool for that workspace.

For other supported launch modes, see [Vitis Unified IDE Launch Options](#).

Note: Before launching the AMD Vitis™ Unified IDE, you can set up other environmental settings to ensure that the tool can pick up these settings. For example, you can set up Xilinx Runtime (XRT) for building and running data center acceleration applications:

```
source <XRT_Install_Path>/setup.sh
```

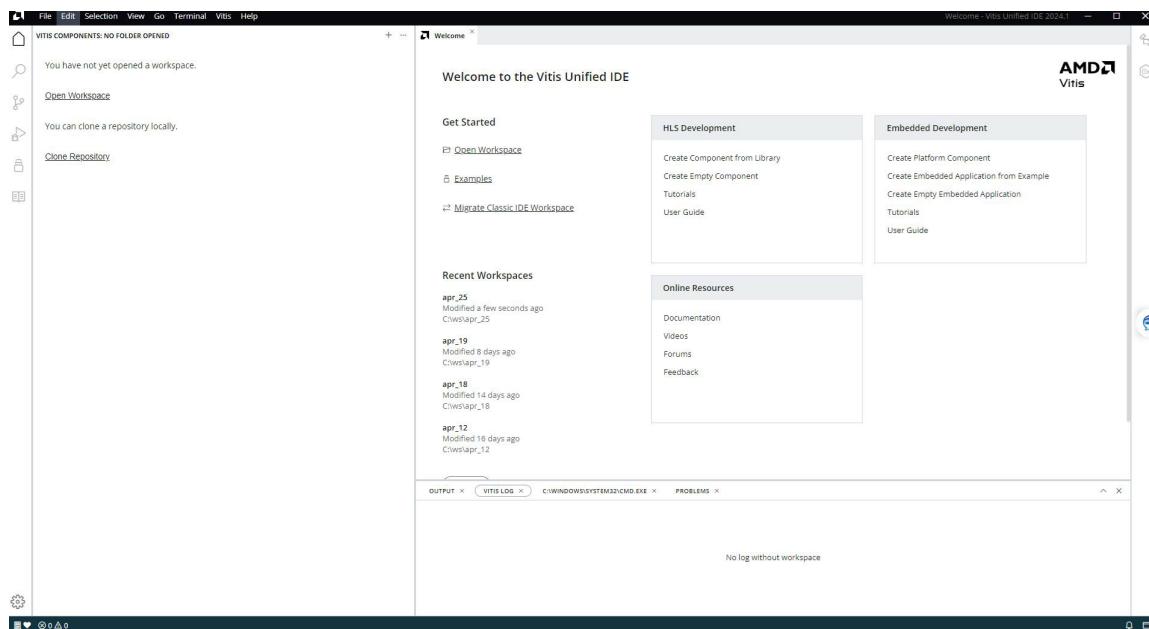
You can set up the platform repository path with the following example:

```
export PLATFORM_REPO_PATHS=<platform_path>
```

The `vitis` command launches the Vitis Unified IDE with your defined options. It provides options for specifying the workspace and options of the project. The following sections describe the `vitis` command options.

The following screen is displayed when the Vitis Unified IDE is launched.

Figure 5: Vitis Unified IDE Welcome Screen



Note: The GUI of this welcome page is different between full installer and Embedded installer.

Vitis Unified IDE Launch Options

Launch Options

The Vitis Unified IDE supports the following modes:

- **GUI Mode:** By default, Vitis Unified IDE launches in GUI mode with a graphical interface. Optionally, you can use the argument `-w` to specify the workspace to open.

```
vitis -w <workspace>
```

- **Analysis Mode:** Analysis mode launches the tool directly into the [Analysis View \(Vitis Analyzer\)](#) letting you review the summary reports generated during the build, run, and debug processes.

```
vitis -a
```

- **Interactive Mode:** Interactive mode lets you enter commands through the interactive Python shell, outside of the GUI, as described in [Vitis Interactive Python Shell](#).

```
vitis -i
```

Note: Type `help()` from the interactive command prompt to explore the available command modules.

- **Batch Mode:** Batch mode executes the specified Python script and exits.

```
vitis -s <script>.py
```

- **Jupyter Notebook Mode:** Launches Jupyter Notebook server with the Vitis environment and the front end UI in your default web browser.

```
vitis -j
```

You can use `vitis_server` command line interface in this environment as described in [Vitis Interactive Python Shell](#).

You can use `-h` to print the supported options of `vitis`.

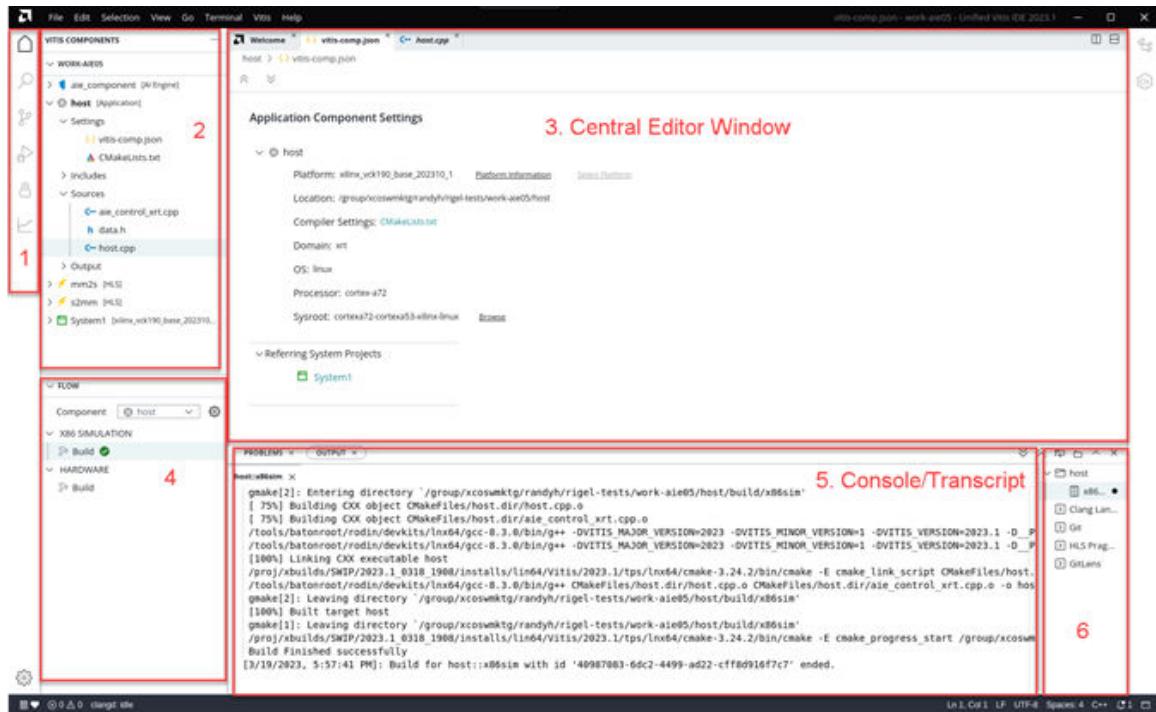
```
vitis -h

Syntax: vitis [--classic | -a | -w | -i | -s | -h | -v]

Options:
  -classic/--classic
    Launches New Vitis IDE (default option).
  -a/--analyze [<summary file | folder | waveform file: *.[wdb|wcfg]>]
    Open the summary file in the Analysis view.
    Opening a folder opens the summary files found in the folder.
    Open the waveform file in a waveform view tab.
    If no file or folder is specified, opens the Analysis view.
  -w/--workspace <workspace_location>
    Launches Vitis IDE with the given workspace location.
  -i/--interactive
    Launches Vitis python interactive shell.
  -s/--source <python_script>
    Runs the given python script.
  -j/--jupyter
    Launches Vitis Jupyter Web UI.
  -h/--help
    Display help message.
  -v/--version
    Display Vitis version.
```

Vitis Unified IDE View and Feature

Figure 6: Vitis Unified IDE

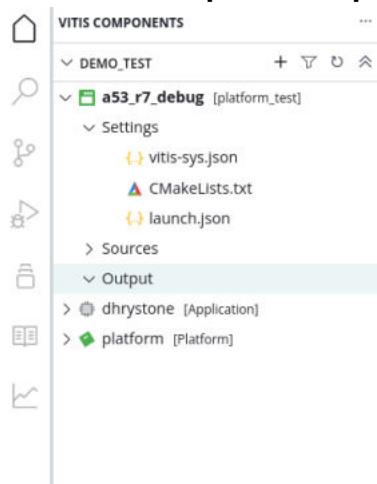


1. The Toolbar menu at the left of the screen provides easy access to major features of the tool: the Vitis Component Explorer, the Search function, Source Control, the Debug view, Examples, and the Analysis view.
2. Toolbar menu can customized based on your preference using View dropdown.
3. The Vitis Component Explorer can be used to view a virtual hierarchy of the workspace. The view displays a workspace that is structured to help you understand the different elements of the component or project, for example a Sources and Outputs folder that does not exist on disk.
4. The Central Editor window is used for editing components, configurations, and source files.
5. The Flow Navigator displays the design flow for the active component. Different components has different work flows, and the work flow of the active component is displayed in the Flow Navigator. You can specify the active component by selecting it in the Flow Navigator, or selecting it in the Component Explorer.

6. The Console/Terminal area displays the output transcripts of the tool, and other windows such as the Terminal window and the Pipeline view are also located here. The terminal window displays the folders of the workspace and can be used to run scripts on the content.
 7. The Index displays a list of transcripts of the various build steps ran during the session and allows you to reopen a process transcript that was closed earlier.
-

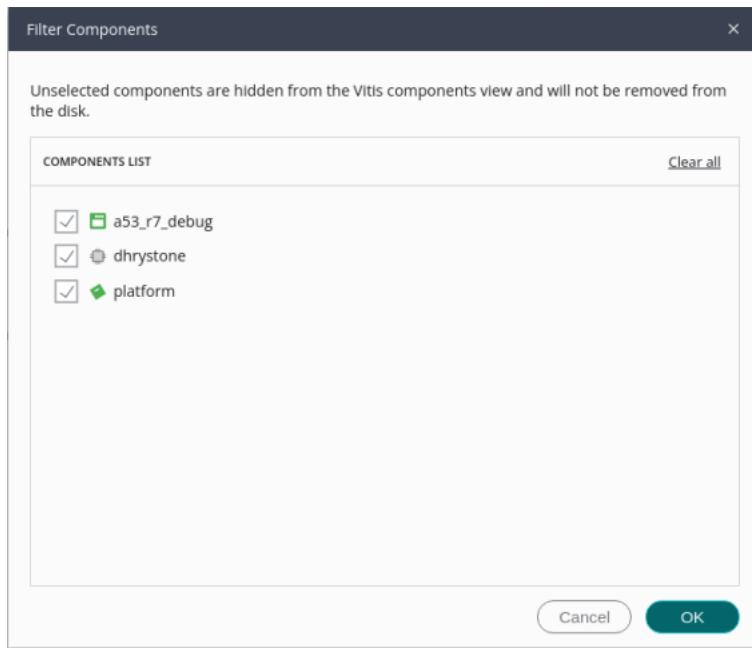
Vitis Component View

Figure 7: Vitis Component Explorer



You can use the Component Explorer to perform the following actions:

- **Via the toolbar menu:**
 - You can create a New Component.
 - You can select Filter Components to hide or show the components of interest.



- You can refresh the Component Explorer.
- You can Collapse All to minimize the displayed content of the components and projects.
- **Via selecting a component or project and using the right-click menu:**
 - You can select **Open in Terminal** to open a terminal window and changes directory to inside the component.
 - You can select **Show in Flow Navigator** and click the component to show it in the Flow Navigator.
 - You can select **Delete** to remove the component or project from the workspace.
 - You can select **Clone Component** to create a new component from an existing component. This is useful for design exploration where you preserve the original component as your baseline, and use the cloned component for design exploration. This command does not support System projects.
 - You can Reset Linker Script to generate a linker script for a standalone application component.
- **Via right-clicking the Sources folder of a component or project:**
 - You can select **New File** to create a new file in the component or project.
 - You can select **New Folder** to create a new folder in the component or project.
 - You can select **Open in Terminal** to open a terminal window and changes directory to inside the component.
 - You can select **Paste** to take a copied item and paste it into the component or project. This creates an actual copy of the previously selected and copied item.

- You can select **Import→Files** to import files into the component or project.
- You can select **Import→Folders** to import a folder into the component or project.
- You can select **Add Source File** or create a New Source File.
- **Via right-clicking an object in the component or project hierarchy:**
 - You can select **Copy** to copy the current object. This action can be used along with **Paste** to copy an object from one component or project to another.
 - You can select **Copy Path** to copy the absolute path of the object. The path can be pasted into the Terminal view of a configuration file.
 - You can select **Copy Relative Path** to copy the path of the object relative to the current workspace. The path can be pasted into the Terminal view of a configuration file.
 - You can select **Delete** to remove the object from the workspace.

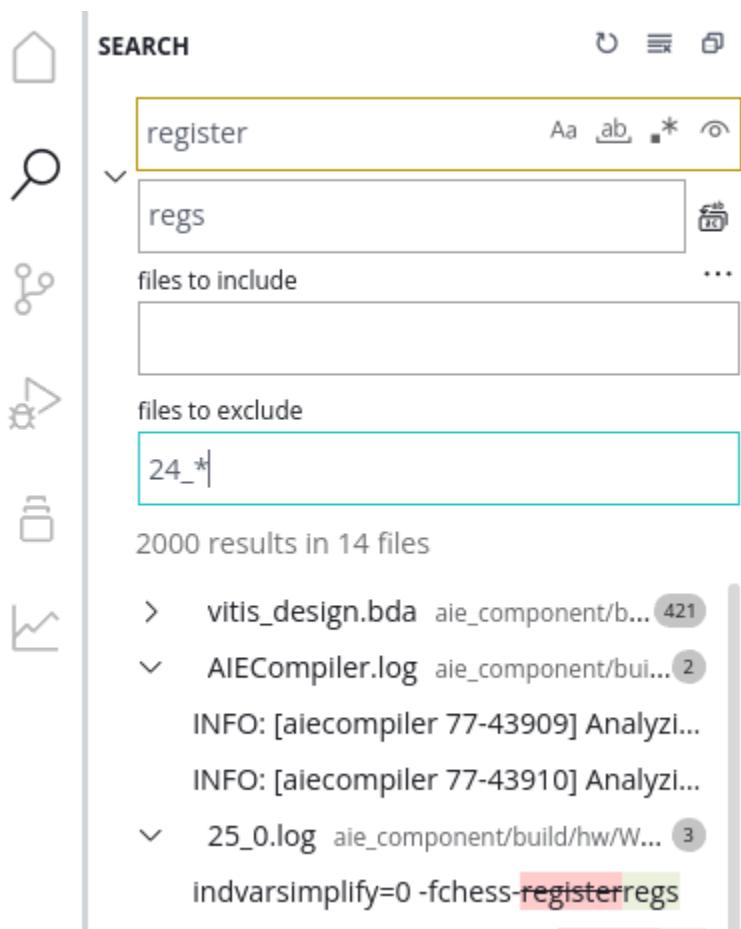
Flow window is a part of Vitis Component View. You need to add a small section to introduce flow.

The Flow window is at the bottom part of the Vitis Components View. When you select a component, the Flow window actively displays the range of actions you can perform on that component, including options like build, debug, and so on.

Search View

The search view can be used to find and replace text globally within the current workspace. The search result includes text files in the workspace including source files, configuration files, and log files. The search is performed inside files. File names are not part of the search.

Figure 8: Search View



As shown in the image above, some of the features of the Search view include the following:

- **Match Case:** Match the case of the provided search string
- **Match Whole Word:** Match only whole words
- **Use Regular Expression:** Use regular expressions to define the search
- **Include Ignored Files:** Restores ignored files to the search list
- **Replace All:** When replace is enabled, replace all search results
- **Toggle Search Details:** Expands the Search view to add Files to Include and Files to Exclude fields
- **Files to include:** Specify a list of files to restrict your search. The listed files can include wildcards, and each entry must be separated by a comma. For example, the following includes (or excludes) the `AIECompiler.log` file and all files with `summary` in the name:

```
AIECompiler.log, *summary*
```

- **Files to exclude:** Specify a list of files to restrict your search. See above for example.
- **Clear Search Results:** Clears the search string and search results



TIP: Be careful when using the *Replace* function, as it can introduce errors into your designs.

Source Control

Source Control or Version Control techniques are widely used in the software development flow. This chapter describes how to use the Git integration in Vitis unified IDE .

Source Control

To enable the Source Control view, you must initialize your empty workspace as a git repository. After creating an empty workspace, and launching the Vitis Unified IDE to open the workspace, you can add it to your git repository using the following steps:

1. From the Terminal menu, select **New Terminal**. The terminal is opened by default to the folder that is your workspace.
2. In the Terminal window, type in the command `git init` and press **Enter**.

You should see a message such as: *Initialized empty Git repository in /tests/temp/workVADD/.git/*.



IMPORTANT! Using the Source Control view with Git requires you to have a User ID and Password established, and provided to the system.

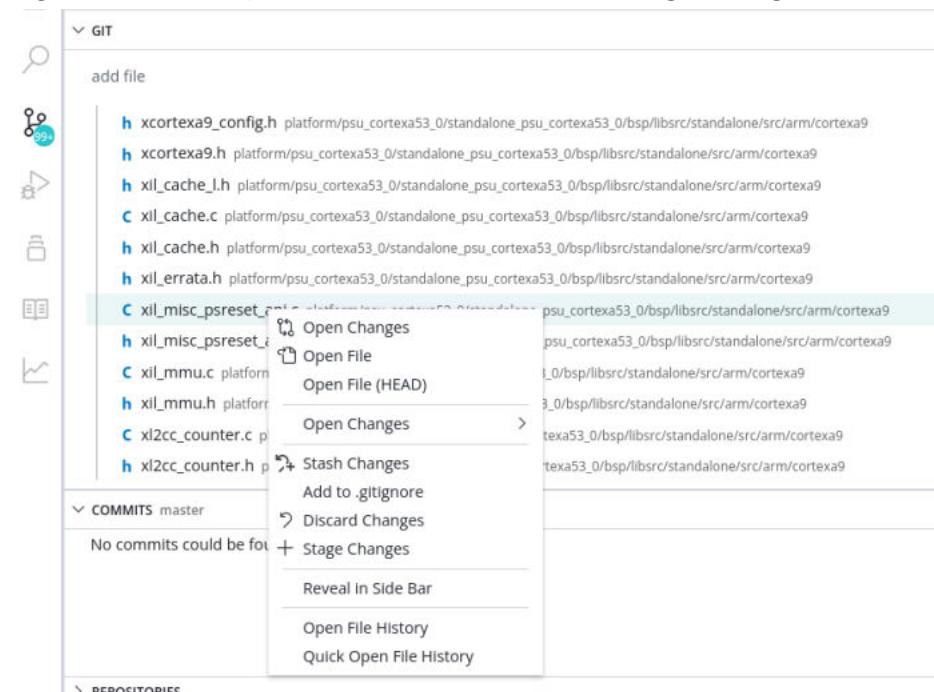
After you create a new component or project in the workspace, Vitis Unified IDE generates a `.gitignore` file. The `.gitignore` file can help you filter out the generated files so that it is easier to pick the files for source control. You can open the `.gitignore` file and edit this file if you have additional requirements.

The Source Control view is a GUI helper for Git. You can use Git commands and the source control view simultaneously for your project. Updates in the command line are displayed in the source control view and vice versa.

Add New File for Source Control

To add a file for source control, you can do the following:

Right click the file you want to add and select **+ Stage Changes**

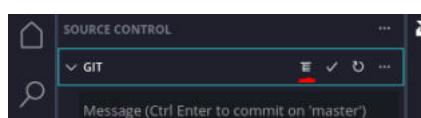


This GUI is equivalent to the following git command.

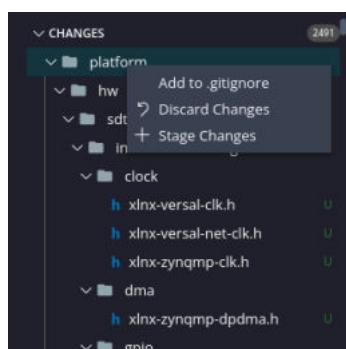
```
git add <file name>
```

Add Components for Source Control

1. Switch to **Source Control** view
2. Switch to view as tree



3. Right click the component you want to add for source control and select **+ Stage Changes**.
 This action will add all the files for this component to do source control.



4. Right click the component and select - **Unstage changes**

This GUI is equivalent to the following git command.

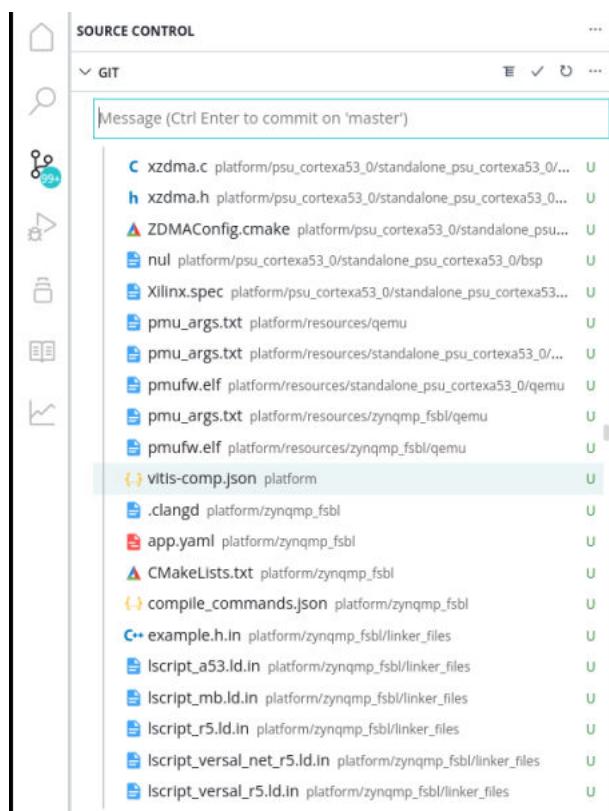
```
git add platform/
```

Note: User need to select the components bound with system project together if they want to add system project for source control.

Commit the Changes

To commit the change, you can do either of the following:

Input the commit message and Press **Ctrl + Enter** after you select the git view.



This GUI is equivalent to the following git command:

```
git commit -m <commit message>
```

Push the Project to the Remote Repository

To push to your remote repository, you can do either of the following:

```
git push --set-upstream origin master
```

- origin: this is the remote repo address
- master: this is the branch of your local workspace code version.

```
git push https://your_repo/vitis_project master
```

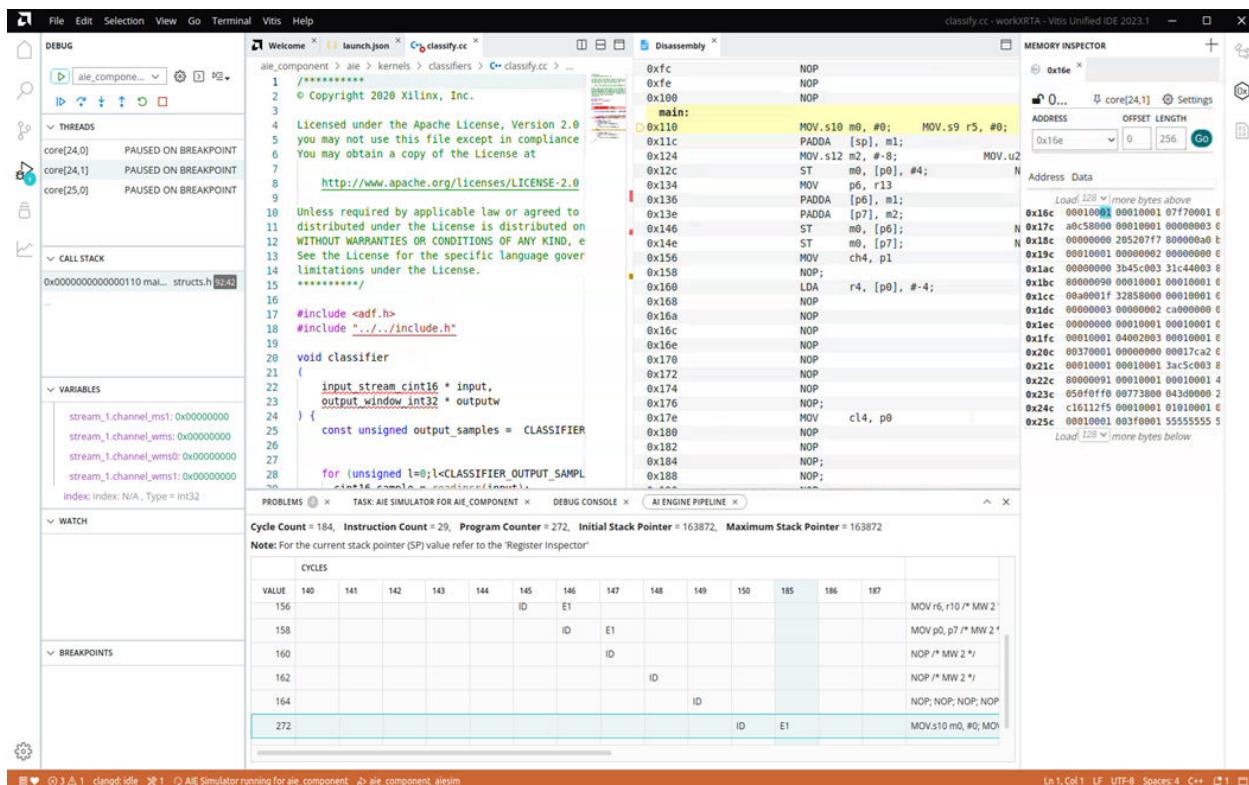
Note: The first time you execute `git init`, it automatically creates a branch named **master**. You can use `git branch <branch name>` to create a new branch.

You can find your local project in the remote repository.

Debug View

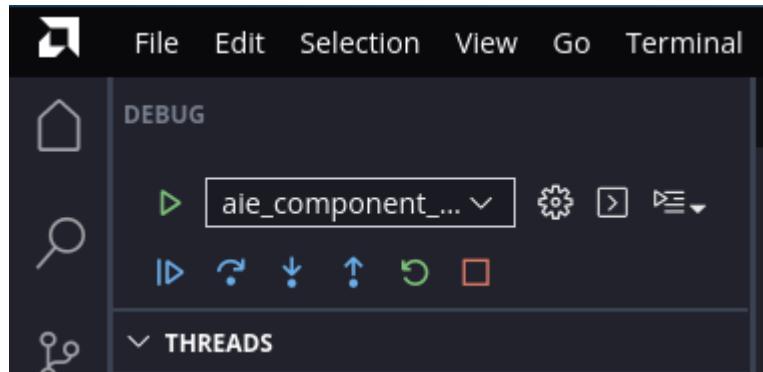
The Vitis IDE debug environment has many features found in traditional GUI-based debug environments, such as GDB. You can add break points to the code, step over or step into specific lines of codes, loops, or functions, and examine the state of variables and force them to specific values. The Debug view contains several windows or views as shown in the following figure.

Figure 9: Debug View



- **Control Panel:** The Debug view's Control Panel is displayed in the upper-left corner of the screen as shown in the image below. During debugging, you can use the control buttons such as Continue, Step Over, Step Into, Step Out, Restart, and Stop to control the debugging process.

Figure 10: Debug Control Panel



- **Threads:** Threads shows the related debugging threads. Threads are created and destroyed during the debugging process. You can switch between multiple threads.
- **Call Stack:** Call Stack shows the function call stack being updated as the application is run.
- **Variables:** Variables shows the current value of global and local variables. When switching threads, the variable information is updated.
- **Watch:** Watch shows variables and expressions you have specified to watch. To add watch points select **Add Expression** (+).
- **Breakpoints:**

Vitis IDE sets break points at the main function of the host component and at the top function of the PL kernels if they can be debugged. To add breakpoints, you can open the source file and click the left side of the line number when a red dot appears. You can remove breakpoints by clicking on a previously added breakpoint.

You can add conditional breakpoints by right-clicking when the red dot appears and select **Add Conditional Breakpoint**. You can also right-click and select **Add Logpoint** to insert a message to be logged when the breakpoint is reached.

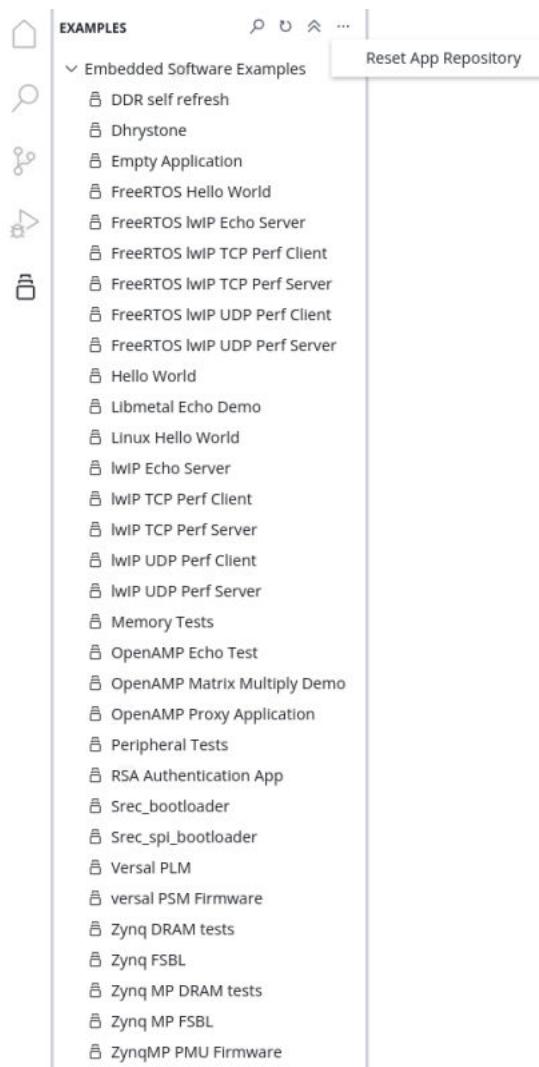
- **Source Code Editor:** The Source Code view is opened when Debug is launched from the Flow Navigator or Launch Configuration view.
- **Memory Inspector:** You can manually open the Memory Inspector as described in [Viewing Memory](#). The Memory Inspector displays the content of specific memory addresses.

- **Register Inspector:** You can manually open the Register Inspector as described in [Viewing Memory](#). The [Viewing Registers](#) shows the registers of the Cortex-A72 when a breakpoint is triggered in the Application Component source code, and the AI Engine when a breakpoint is triggered in the AI Engine kernel.
 - **Disassembly View:** You can open the Disassembly view from the Source Code window right-click menu.
 - **Debug Console:** Displays the transcript of the debug process, and any messages received from the tested application.
-

Example View

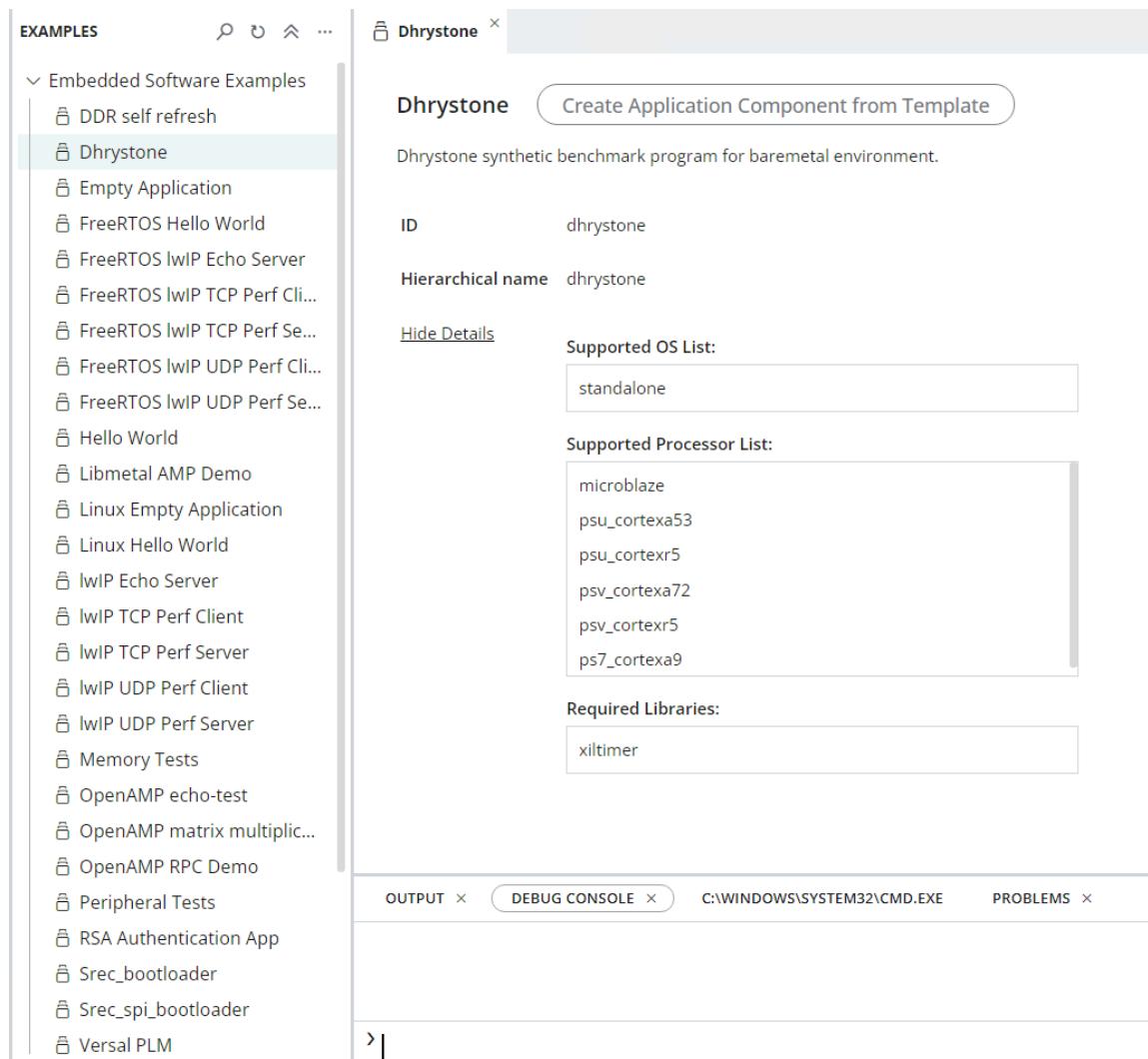
You can access the Examples View from the left side panel or from **View → Examples**, or by using **Ctrl + Shift + R** shortcut keys. In this view, you can create example System projects to explore the tool, and manage example repositories.

Figure 11: Examples View



To use an example, select it from the Examples view and a template opens to let you create a new application component. Click **Show Details** to display the supported OS and processor for this application.

Figure 12: Example View Supported Devices



Code View and Smart Editor

The Source Code editor in the Vitis Unified IDE supports the following features:

- Syntax highlight for C, C++, Python, Makefile, CMakeList.txt file
- Hint for variable names, function names, etc
- Jump to the definition of variables or functions
- Peek the definition of variables or functions so that you need not leave the editing file
- Report the references of variables and function

You can open the outline window by selecting the button  on the right, or selecting the Outline View from the View menu. The Outline window can list the function names in your source code.

The smart editor for `launch.json` files and `build.json` files can switch views between GUI rendering, Text rendering and Text Editor view with the table button  and code button . The GUI rendering only displays the options that it can recognize for the context. For advanced use cases, you need to edit the configuration file manually. The modifications in one view updates the other view instantly. The following figures are GUI format and text format.

Figure 13: **GUI Format**

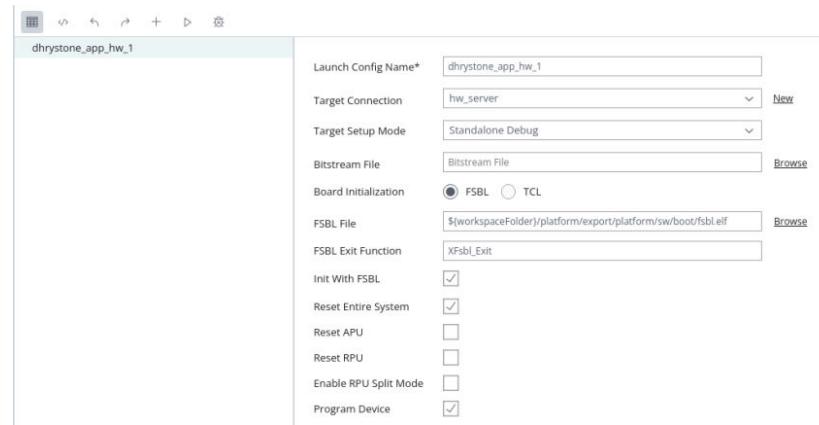


Figure 14: **Text Format**

```

1  {
2      "version": "0.2.0",
3      "configurations": [
4          {
5              "type": "tcf-debug",
6              "request": "launch",
7              "name": "dhrystone_app_hw_1",
8              "debugType": "baremetal-zu",
9              "attachToRunningTargetOptions": {
10                  "targetSetupMode": "standalone",
11                  "executeScript": true,
12                  "scriptPath": ""
13              },
14              "autoAttachProcessChildren": false,
15              "target": {
16                  "targetConnectionId": "hw_server",
17                  "peersIniPath": "../../../.peers.ini",
18                  "context": "ZUPlus"
19              },
20              "targetSetup": {
21                  "resetSystem": true,
22                  "programDevice": true,
23                  "enableRPUSelectionMode": false,
24                  "resetAPU": false,
25                  "resetRPU": false,
26                  "bitstreamFile": "",
27                  "zuInitialization": {
28                      "zuInitialization": true
29                  }
30              }
31          }
32      ]
33  }
  
```

The changes are saved automatically when Auto Save is enabled. Optionally, you can manually save changes in a file with keyboard shortcut `Ctrl + S`.

Preferences

The **File → Preferences** menu leads to a variety of user preferences that can be set and maintained to configure the Vitis Unified IDE. The following are items that are associated with the Preferences menu and command.

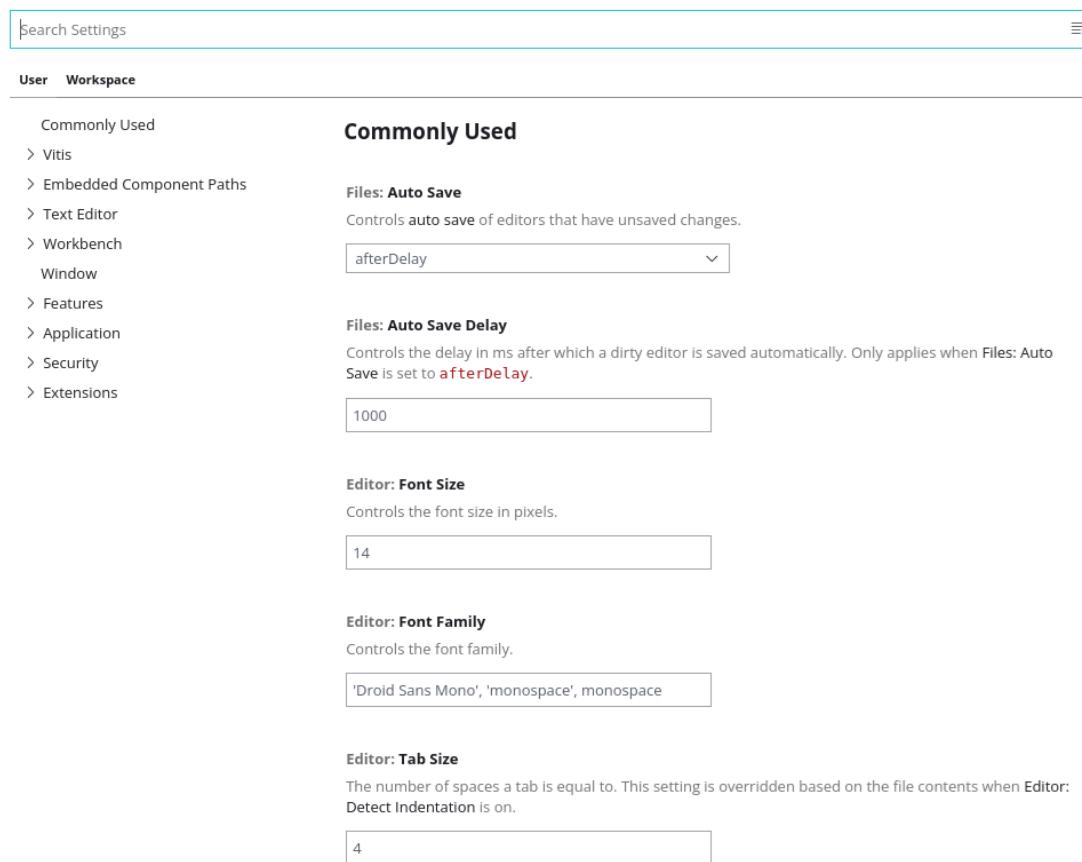
Additionally, From button below could also go to **Preference**.



Settings

- **Auto Save:** The Auto Save command on the File menu is enabled by default to allow the tool to save your changes to configuration files, source code, `CMakeLists.txt` files and more. Auto Save triggers and Auto Save Delay values can be defined from the Preferences page.
- **Color Themes:** Specify a Light Theme or a Dark Theme for the Vitis Unified IDE according to your preferences, or available lighting.
- **Open Settings (UI):** Displays the Preferences view, which has a number of settings as indicated by a Table-of-Contents on the left hand side, and a series of options available for you to configure the tool on the right-hand side. The Preferences page presents a wide array of options to configure your design environment, starting with general options like font styles and sizes to configure the environment to your tastes and needs.

Figure 15: Common Preferences



- **Open Keyboard Shortcuts:** Define new keyboard shortcuts for the various commands of the tool, as described in [Keyboard Shortcuts, Command Palette, and Quick Find](#).
- **File Icon Theme:** Specify a file icon theme to be used in your environment.

Note: You can press **Ctrl + +** to zoom into the display, or press **Ctrl - -** to zoom out the display.

Keyboard Shortcuts, Command Palette, and Quick Find

Keyboard Shortcuts

You can review and edit keyboard shortcuts from **File → Preferences → Open Keyboard Shortcuts**, or using the shortcut key **Alt + Ctrl + Comma**. For example:

1. Use **Alt + Ctrl + Comma** shortcut to open the Keyboard Shortcuts window.
2. Type **build** in the search bar. Matching keyboard shortcuts are displayed.
3. The Build: Hardware key-binding is **Alt+Shift+BH**.

4. To edit the keyboard shortcut for build hardware, hover over the line of Build: Hardware, click the **Edit Keybinding** icon on the left and input your preferred key-binding in the pop-up window.

Command Palette

The command palette lets you quickly find and execute commands without remembering the keyboard shortcuts or menu location. For example, to run build hardware with the command palette, do the following:

1. Select your component or project in the workspace.
2. Type **Ctrl + Shift + P** to open the Command Palette menu.
3. Type in the keyword `build` or `run` or `debug` for example, and the Command Palette filters the list of command names until it includes only those commands that match your text entry.
4. Use the mouse, or the up or down arrow keys to scroll through the selection of commands and select the command you want to execute.
5. Press **Enter** to execute the selected command.

For example, select an HLS component in the Component Explorer and type **Ctrl + Shift + P**, then in the Command Palette type `C Syn`. The HLS: C Synthesis is displayed as one of the options. Select the option and press **Enter**. The tool runs C synthesis on the selected HLS component.

Quick Find

Click **Ctrl + P** to open the Quick Find box. Type in a file name and the tool begins to find files that match your search string. Select a file and hit **enter** to open it.

With a file open, you can type `@` in the Quick Find box to go to a function within the opened code, or type `:40` for example, to go to line 40 of the file. You can use this method to quickly find and jump to a function or line number.

With a file open, you can type `#` in the Quick Find box to go to a function within the workspace. You can use this method to quickly find and jump to a function.

Click **Ctrl + T** to open the Quick Find box. Type the symbol name and the tool begins to find the symbol that matches your search string. Additionally, it searches the files.

Parallel Compiling

The Vitis Unified IDE provides fast responses to actions. It employs non-blocking build commands that allow you continue working and run multiple builds at the same time. If your system is sufficiently powerful, you can launch multiple actions together, that is the build software emulation and hardware emulation, or debug software emulation when building hardware emulation is still running.

Note: If your server is not powerful enough, it's possible to see a lag and slow response from the Vitis Unified IDE if you launch multiple build or emulation jobs at the same time.

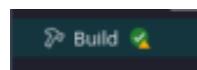
The application building job can also launch multiple components in parallel. Building jobs for the referenced components can be launched in parallel.

The Parallel Build feature is disabled by default. You can enable it by **File → Preferences → Open Settings (UI)** command and selecting the **Vitis → Build → Parallel Build → Disable** setting.

Notification for File Change

When source code or setting files are modified, a yellow indicator beside the build status icon appears, signaling the user to review the status and make a decision to rebuild the component.

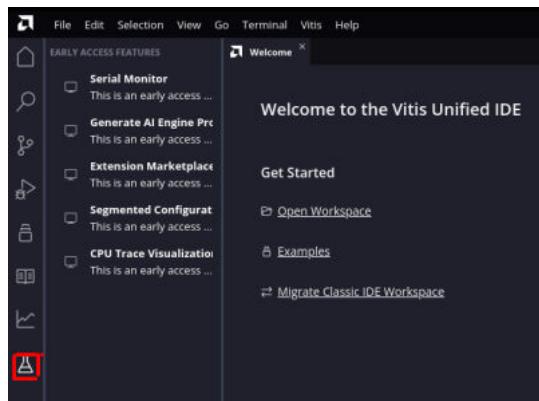
Figure 16: Out-of-date notification for components



New Feature Preview

New features are introduced for you to explore in advance. These features are functional and have undergone preliminary testing; however, they might not address all potential use cases and might require further development before they can be universally implemented. Share your feedback and insight. You can view the new features by clicking **Vitis → New Feature Preview**. The following page and icon appears.

Figure 17: New feature preview



A funnel-shaped icon appears in the left view bar allowing you to access the new features. You can click on it to access new features and to view more information about the feature.

Disclaimer: The previewed features are currently in the early access phase. They are currently being development and might not be fully functional or stable. By accessing and using these features, you acknowledge and accept the following:

- Limited functionality: Early access features might have limited functionality compared to fully released features. They can lack certain functions, have bugs, or experience performance issues. Be informed that your experience with these features might not be optimal.
- Potential instability: Early access features are still being tested and refined. As a result, they are prone to crashes, errors, or unexpected behavior. Use these features with caution and understand that they might not always work as intended.
- Feedback and improvements: Your feedback is crucial in improving early access features. AMD encourages you to report any issues, bugs, or suggestions you encounter while using these features. Your input helps AMD to enhance their performance and stability.
- No guarantees: Early access features are provided on an "as-is" basis, without any warranties or guarantees of any kind, whether expressed or implied. AMD does not guarantee that these features are released in their current form or at all. AMD reserves the right to modify, suspend, or discontinue these features without prior notice.
- Use at your own risk: By using early access features, you understand and accept the risks involved. AMD shall not be held liable for any damages, losses, or inconveniences arising from the use of these features.



IMPORTANT! Carefully consider these factors before accessing and using early access features. Your participation in testing and providing feedback is greatly appreciated as it improves and shape these features for a better user experience.

Develop

This section describes how you can use the AMD Vitis™ integrated design environment (IDE) to create and manage target platforms and applications.

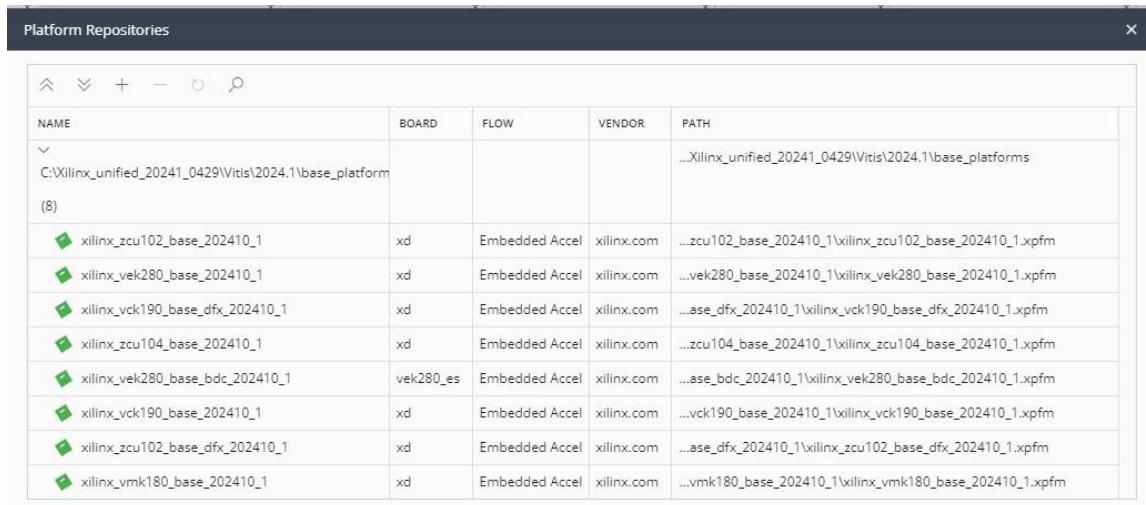
Managing Platforms and Platform Repositories

The Platform Repositories window displays the platforms that are available for use with the new Vitis unified IDE. The upper part of the table displays the base platforms that are installed with the software installation, and are available to all users. The lower part of the table displays the locations specific as part of the `$PLATFORM_REPO_PATHS` environment variable. This shows the platforms that are installed in addition to the Vitis tools to work with the tools.

You can manage the platforms that are available for use by going to **Vitis → Platform Repositories** in the main menu of an open project.

1. Click + or - icons to add or remove a platform search directory from the list.
2. Select a platform directory to view platforms of that directory.
3. Select the **info** link next to a platform to view its detailed information.

Figure 18: Platform Repositories



NAME	BOARD	FLOW	VENDOR	PATH
C:\Xilinx_unified_20241_0429\Vitis\2024.1\base_platform (8)				...Xilinx_unified_20241_0429\Vitis\2024.1\base_platforms
xilinx_zcu102_base_202410_1	xd	Embedded Accel	xilinx.com	...zcu102_base_202410_1\xilinx_zcu102_base_202410_1.xpfm
xilinx_vey280_base_202410_1	xd	Embedded Accel	xilinx.com	...vey280_base_202410_1\xilinx_vey280_base_202410_1.xpfm
xilinx_vck190_base_dfx_202410_1	xd	Embedded Accel	xilinx.com	...ase_dfx_202410_1\xilinx_vck190_base_dfx_202410_1.xpfm
xilinx_zcu104_base_202410_1	xd	Embedded Accel	xilinx.com	...zcu104_base_202410_1\xilinx_zcu104_base_202410_1.xpfm
xilinx_vey280_base_bdc_202410_1	vey280_es	Embedded Accel	xilinx.com	...ase_bdc_202410_1\xilinx_vey280_base_bdc_202410_1.xpfm
xilinx_vck190_base_202410_1	xd	Embedded Accel	xilinx.com	...vck190_base_202410_1\xilinx_vck190_base_202410_1.xpfm
xilinx_zcu102_base_dfx_202410_1	xd	Embedded Accel	xilinx.com	...ase_dfx_202410_1\xilinx_zcu102_base_dfx_202410_1.xpfm
xilinx_vmk180_base_202410_1	xd	Embedded Accel	xilinx.com	...vmk180_base_202410_1\xilinx_vmk180_base_202410_1.xpfm

Target Platform

In the Vitis unified software platform, runtime environment of the application is referred to as the *target platform*. A target platform is a combination of hardware components (XSA) and software components (domains, boot components like FSBL or PLM, and so on).

A platform project is a customizable target platform in a workspace. You can add, modify, or remove domains in a platform project. You can also enable, disable, and modify boot components. A domain is referred as a BSP or an OS, which targets one processor or a cluster of isomorphism processors (for example, a 4x Cortex®-A53 cluster with SMP Linux). A platform can contain unlimited domains.

This section explains how to create a hardware design, and how to use that hardware design to create an application platform.

Creating a Hardware Design (XSA File)

AMD hardware designs are created with the AMD Vivado™ Design Suite, and can be exported in the Xilinx support archive (XSA) proprietary file format that can be used by the Vitis software platform. For information on how to create an embedded design in Vivado and generate the XSA file, see the following embedded design tutorials:

- *Zynq 7000 SoC: Embedded Design Tutorial* ([UG1165](#))
- *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* ([UG1209](#))
- *Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* ([UG1305](#))

The generic steps are as follows:

1. Create a Vivado project.
2. Create a block design.
3. Generate the image or bitstream.
4. Export the hardware using **File→Export→Export Hardware**, and select the **Fixed Platform** option.

Creating a Platform Component from XSA

To create a new platform component in the Vitis Unified integrated design environment (IDE), execute the following steps.

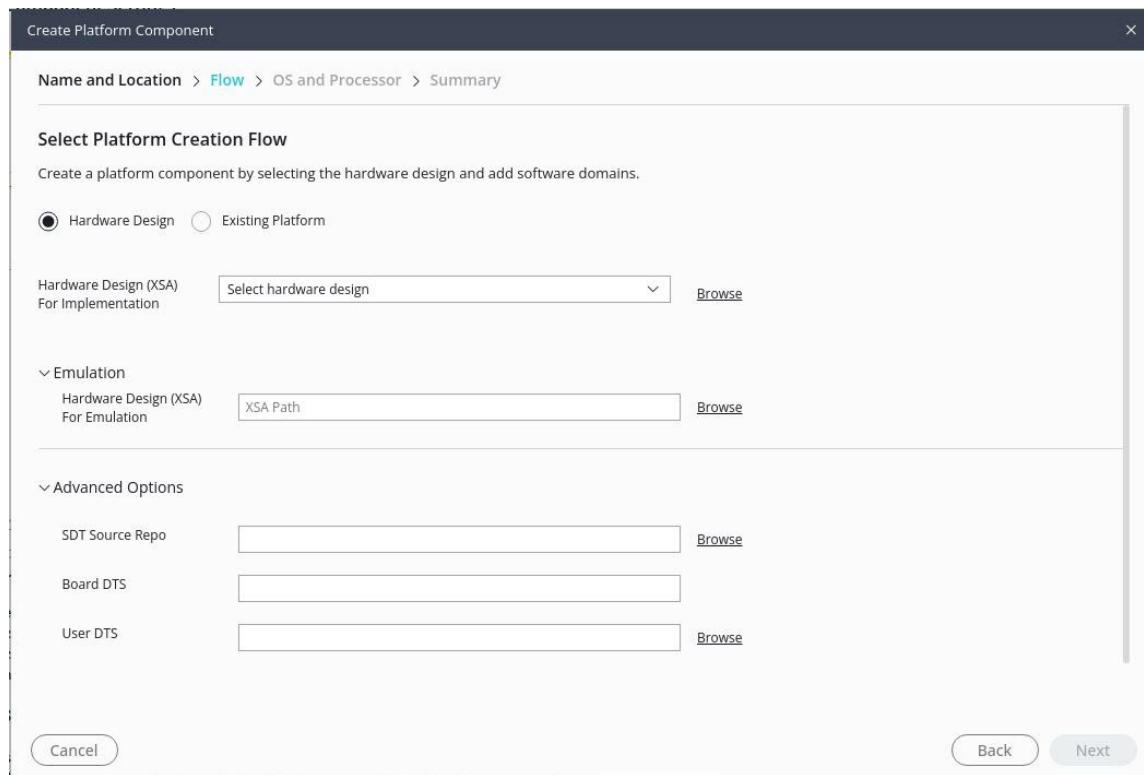
1. Click the File option in the Vitis Unified IDE and select **New Component→Platform**.



TIP: You can also select the **Create Platform Component** command from the Welcome page.

2. This opens the Create Platform Component wizard.

- Enter a Component name and Component location and select **Next**.
- Select **Browse** to locate an XSA file or select to create a platform from existing platform, or use the drop down menu to select the built-in XSA files. The built-in fixed XSA files only contains PS initialization. Click **Next**.



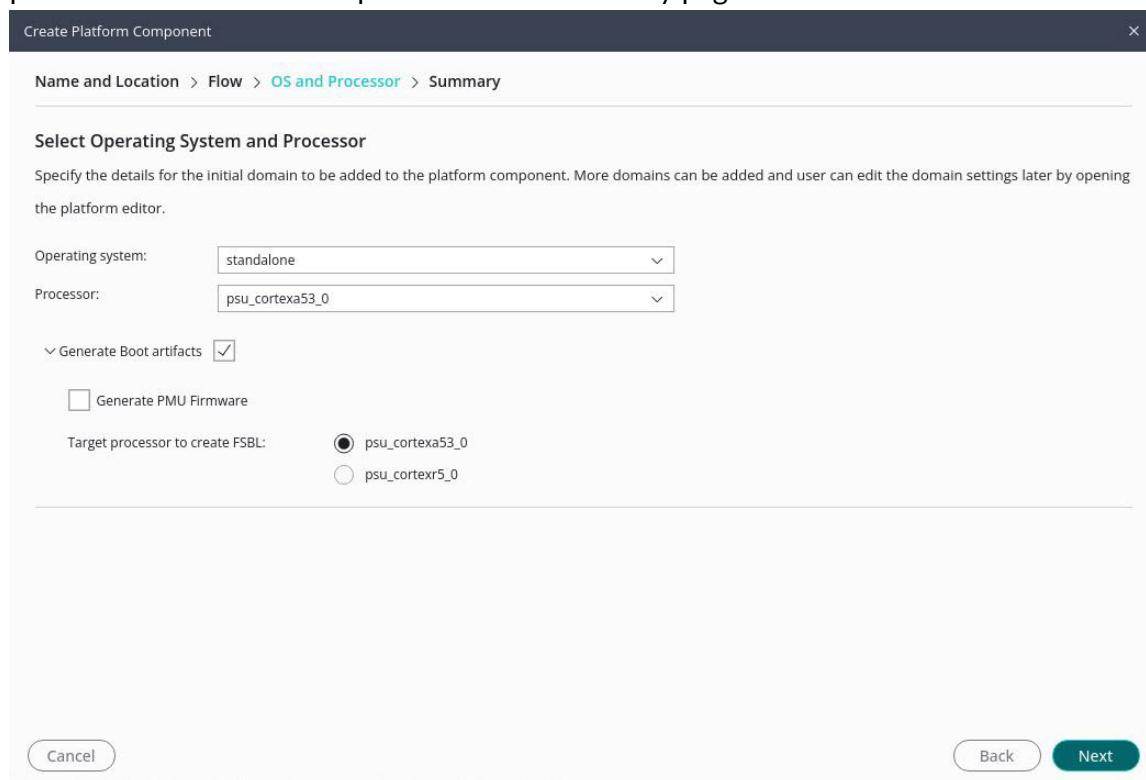
Note: If you select to create a platform from an existing platform, it copies the platform you specified to your current workspace.

Note: If you want to support emulation, expand **Emulation** and select **Browse** to locate an emulation XSA file.

Note: The created platform type is determined by the input hardware design type. If the input hardware design is a fixed XSA, a platform for an embedded design is created. If the input hardware design is an extensible XSA, the created platform is extensible platform.

Note: SDTgen is a built-in tool within the Vitis Unified IDE. It is responsible for reading the XSA file and generating the SDT (system device tree) files. Another built-in tool, Lopper, then parses the SDT and generates the corresponding BSP (Board Support Package) file. To facilitate user debugging of this process and to take advantage of the SDT flow, Advanced options have been introduced. These options allow you to switch to a custom version SDTgen tool. There are also two options to add board-level and user-defined DTS (Device Tree Source) files.

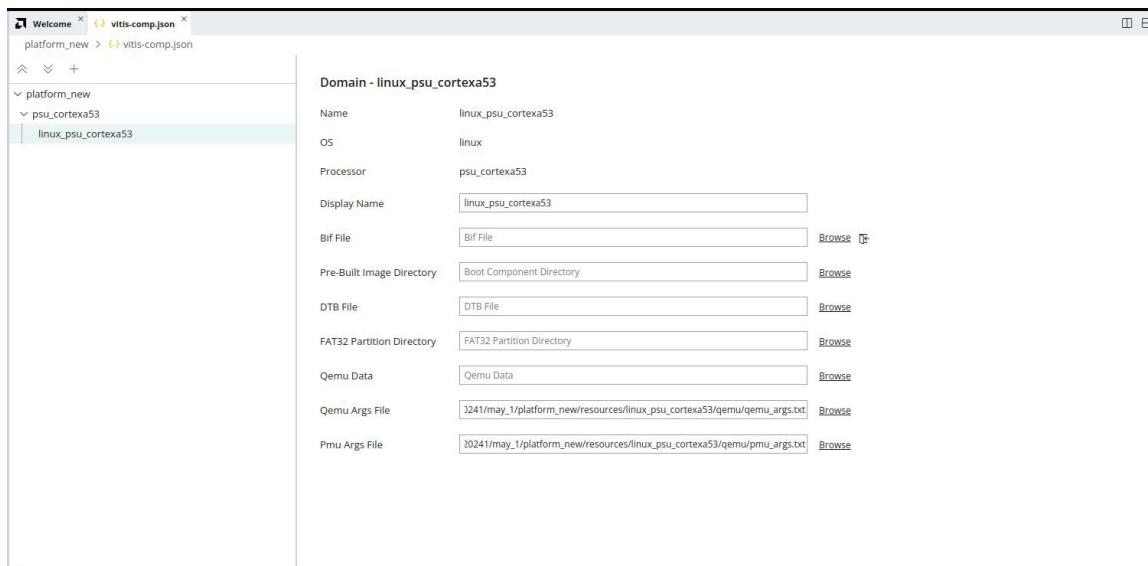
- After selecting the XSA, the tool reads the XSA and identifies the available processors and operating system domains. Specify the Operating system, and the Processor for the platform. and click **Next** to proceed to the Summary page.



Note: When you enable the **Generate Boot Artifacts** and **Generate PMU Firmware** options, the tool automatically generates both the FSBL (First Stage Boot Loader) and PMU (Platform Management Unit) firmware components required for your platform.

- The Summary page reflects the choices you have made on the prior pages. Review the summary and select **Finish** to create the Platform component, or select **Back** to return to earlier pages and change your selections.

When the Platform component is created the `vitis-comp.json` file for the component is opened in the central editor window as shown for the Linux platform below.



Depending on the OS you selected for your platform, and possibly the processor you chose as well, the contents of the platform `vitis-comp.json` can vary.

- For Linux Operating System: As shown above, you need to specify the Bif file, Boot Component Directory, SD Card Directory and as well as the Qemu data. The Qemu Args file are auto-populated by tool.
- For standalone (baremetal) OS: No special operation is required.
- For FreeRTOS: No special operation is required.

After setting, you can build the Platform component by selecting it in the Flow Navigator and selecting the **Build** command. After compilation is completed, the tool populates the platform and its related software and hardware components in the `Output` folder of the Platform component in the Component Explorer.

Customizing a Pre-Built Platform

Platforms are only editable when it is in the workspace. To customize a pre-built platform, import it to the workspace.

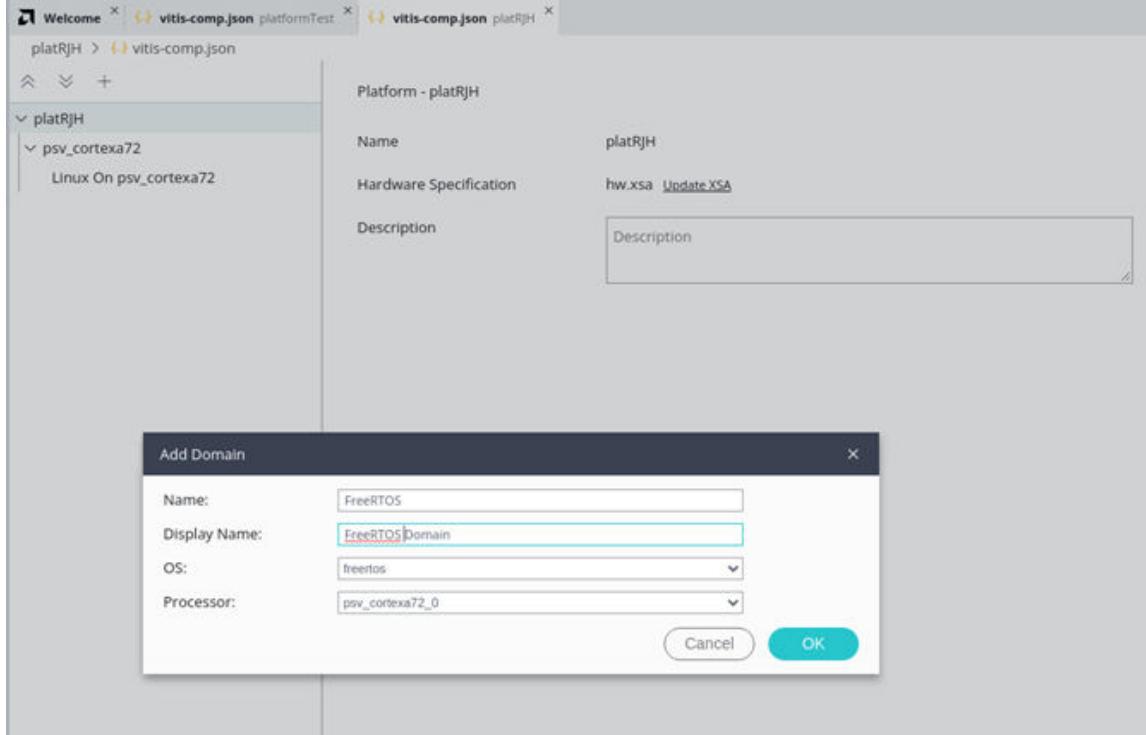
1. Make sure the platform you wish to customize is visible in the platform repository list. If it is not, add the platform path to the platform repository.
2. Launch the New Component wizard using either method below:
 - a. Select **File**→**New Component**→**Platform**.
 - b. From the Welcome page, click **Create Platform Component** to open the New Platform Component wizard.

3. In the New Platform Component wizard, provide a component name and component location in the Component Name and Component Location name field.
4. Click **Next**.
5. In the Platform Component page, select **Existing platform**. Select the desired platform and click **Next** to review the platform component summary.
Note: Make sure the platform to customize is visible in the platform repository list. If it is not, click **+** to add the platform path to the platform repository.
6. Click **Finish**. You can now modify the new platform in the workspace as any other platform.

Adding a Domain to an Existing Platform

A platform can contain multiple domains, supporting different operating systems and targeting different processors. Three types of domains are currently supported: Linux, FreeRTOS, and Standalone (or Baremetal).

1. In the current workspace, expand the Platform component in the Component Explorer to open the Settings folder and select the `vitis-comp.json` file.
Note: To create a Platform component refer to [Creating a Platform Component from XSA](#).
2. Click the **+** button to Add Domain.



3. Specify the Name and Display name.
4. For the OS select Linux, FreeRTOS, Standalone.

5. Select from the available Processors. The selection of Processor changes based on the selected OS.
6. Click **OK** to add the domain to the Platform.

For a FreeRTOS and Standalone domains a Board Support Package (or BSP) is created for the domain. You can specify the libraries to include in the BSP.

For a Linux domain you can configure additional details of the Linux domain by selecting the new domain in the platform. The Boot Components Directory must contain all the components required by the BIF. These components can be generated by PetaLinux.



TIP: The components specified in the Linux domain settings are copied to the platform folder when generating the platform. Adding sysroot to a Linux domain is not supported because Windows does not support copying symbol links.

Configuring a Domain

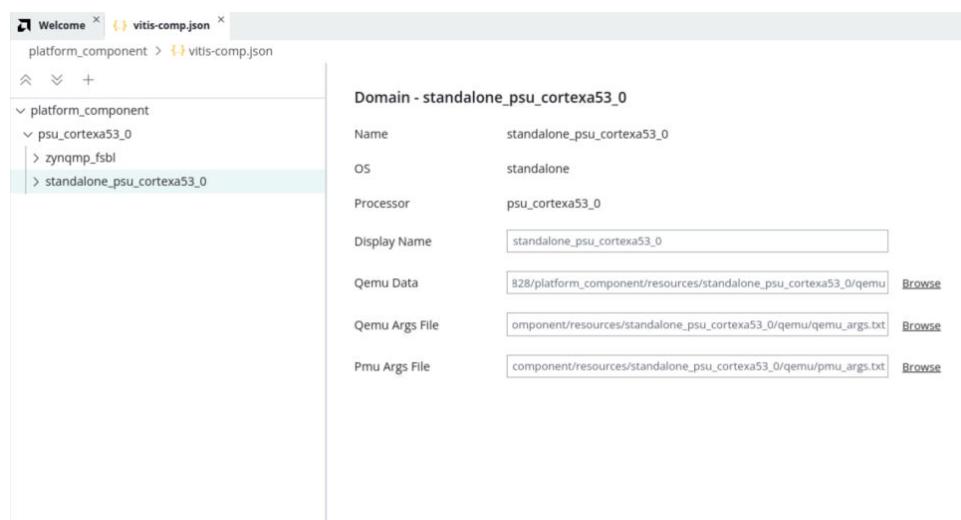
There are different kinds of domains, the standalone domain being the most frequently used. Each domain has an associated BSP which can be configured extensively. Additionally, the domain overview page includes extra settings for the domain.

Domain Overview Page

Standalone and FreeRTOS Domain

The standalone and FreeRTOS domain overview pages are identical and provide a small number of configuration options related to the QEMU emulation platform. These options are auto populated with pre-defined installation file paths.

Figure 19: Standalone Domain Overview Page

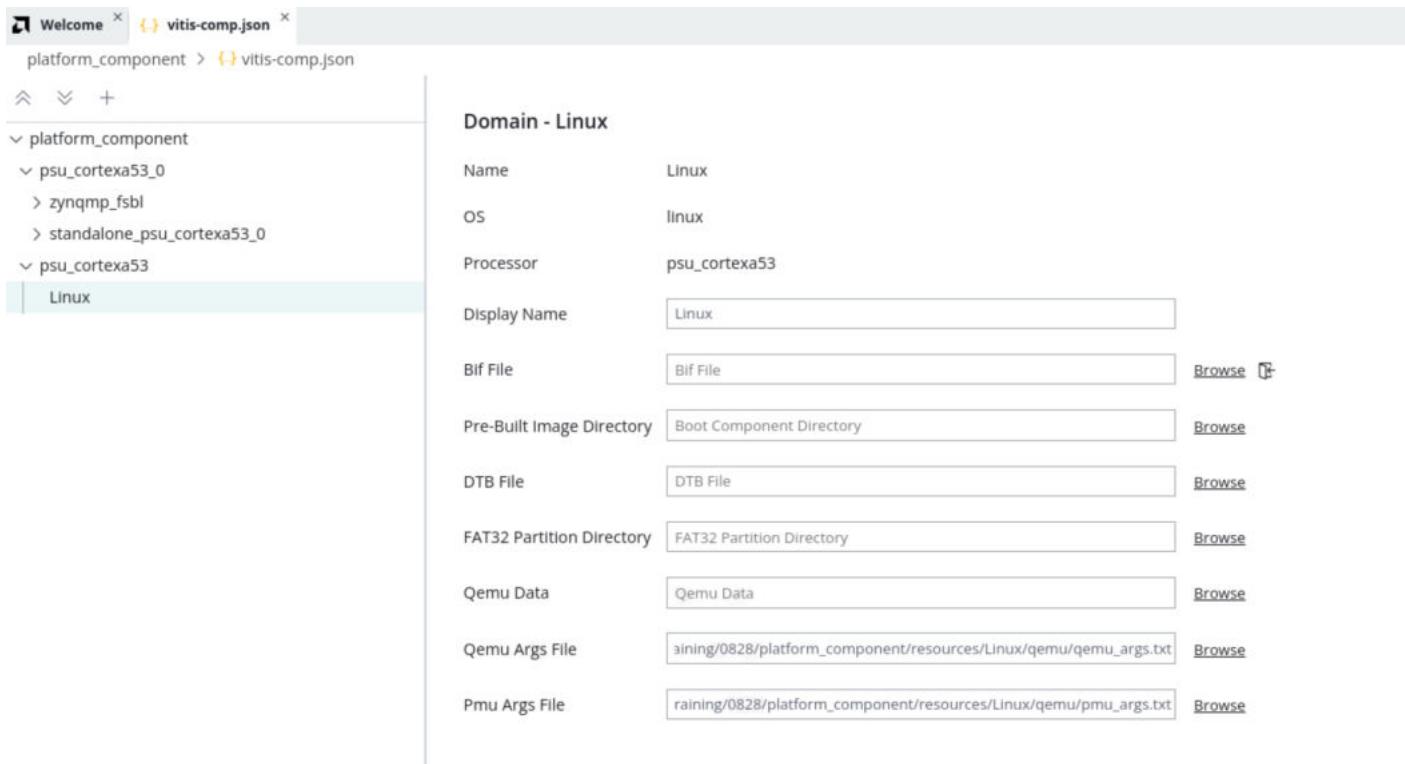


Linux Domain

The Linux domain overview page is similar to the standalone page, but includes more configuration options.

Note: For a fixed platform for embedded application, the following fields do not apply.

Figure 20: Linux Domain Overview Page



- **BIF File:** Boot Image Format file. Click **Browse** to select a `bif` file from the file system, or click the button beside `Browse` to generate the `bif` file for your platform.
- **Pre-Built Image Directory:** Directory containing the Linux image files, including boot components, kernel image and rootfs.
- **DTB File:** The system device tree binary file for Linux system booting. If the Pre-Built Image Directory contains the DTB file, it is automatically populated and displayed in this field.
- **FAT32 Partition Directory:** Used to add additional files to the FAT32 partition.
- **Qemu Data:** Automatically populated when building the platform. This field provides the boot components for hardware emulation.

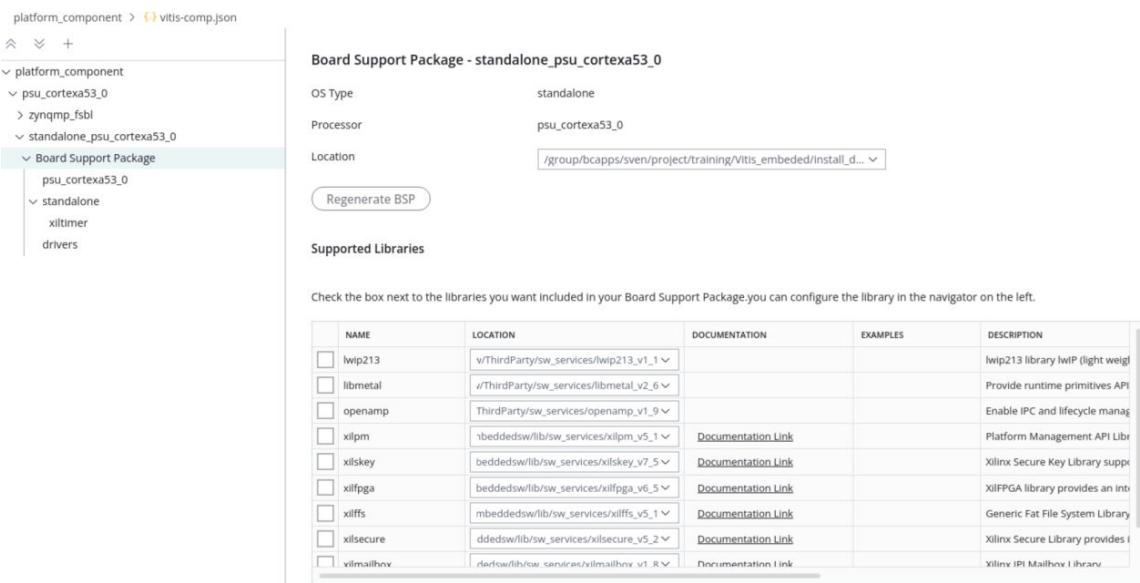
Board Support Package Settings Page

The Board Support Package Settings page includes several configuration pages, and is only applicable for non-Linux domains.

Note: You cannot change the OS choice on this page. The OS type is determined during software platform creation.

Select the platform component in the Component view and click the `vitis-comp.json` file to open it. Next, expand `standalone_psu_cortexa53_0` and click **Board Support Package**. The following configuration page is displayed.

Figure 21: Overview

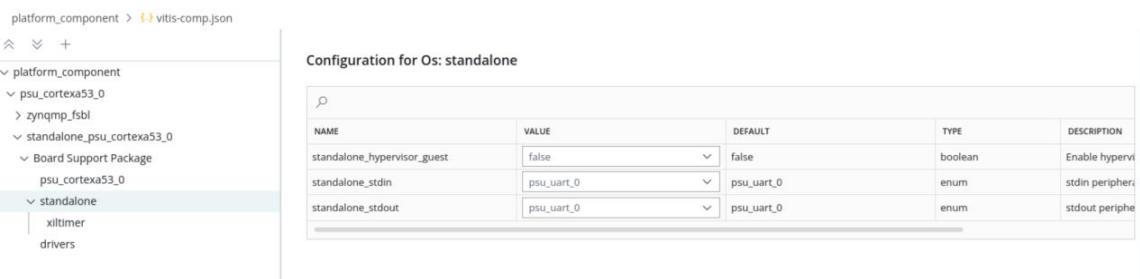


The OS settings section allows you to configure the parameters of the OS.

Figure 22: Processor Parameter Configuration

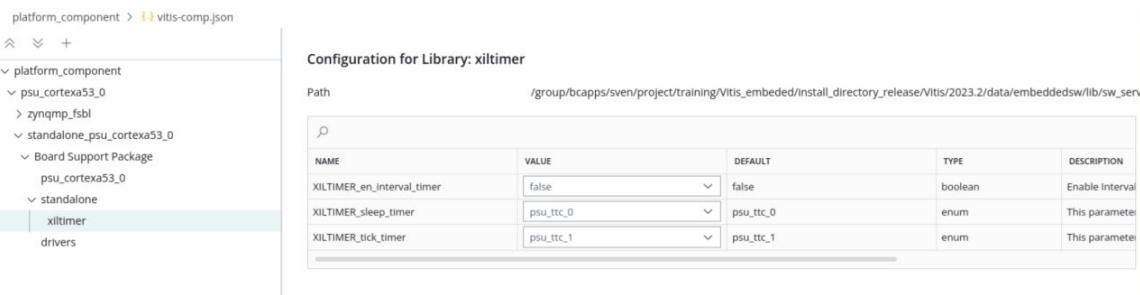


Figure 23: OS Parameter Configuration



The library settings page enables you to configure the parameters of each library enabled in the library page.

Figure 24: Library Configuration



The drivers section lists all the device drivers assigned for each peripheral in your system. You can select each peripheral and change its default device driver assignment and its version. To remove a driver for a peripheral, assign the driver to none.

Figure 25: Drivers

IP INSTANCE	IP TYPE	DRIVER	PATH	DOCUMENTATION
psu_acpu_gic	psu_acpu_gic	scugic	linxProcessorIPLib/drivers/scugic_v5_2	Documentation Link
psu_adma_0	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_1	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_2	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_3	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_4	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_5	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_6	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_adma_7	psu_adma	zdma	linxProcessorIPLib/drivers/zdma_v1_17	Documentation Link
psu_ams	psu_ams	sysmonpsu	rocessorIPLib/drivers/sysmonpsu_v2_9	Documentation Link
psu_apm_0	psu_apm	axipmon	ProcessorIPLib/drivers/axipmon_v6_10	Documentation Link
psu_apm_1	psu_apm	axipmon	ProcessorIPLib/drivers/axipmon_v6_10	Documentation Link
psu_apm_2	psu_apm	axipmon	ProcessorIPLib/drivers/axipmon_v6_10	Documentation Link
psu_apm_5	psu_apm	axipmon	ProcessorIPLib/drivers/axipmon_v6_10	Documentation Link
psu_can_1	psu_can	canps	linxProcessorIPLib/drivers/canps_v3_7	Documentation Link

The build settings section lists the toolchain selected to build the BSP as well as some extra configuration settings.

Switching FSBL Targeting Processor

You can select the target processor for FSBL when creating the platform. After creating the platform, you can re-target it to another processor on a Zynq UltraScale+ MPSoC device. To re-target the platform component to Cortex-R5F, follow the steps below.

1. Click your platform component, expand Settings and double click the **vitis-comp.json** file.
2. Select **psu_cortexa53_0** → **zynqmp_fsbl**.
3. Click **Re-target to psu_cortexr5_0**.
4. Rebuild the platform.

Domain - zynqmp_fsbl

Name	zynqmp_fsbl
OS	standalone
Processor	psu_cortexa53_0
Display Name	zynqmp_fsbl

FSBL is currently targeted to 'psu_cortexa53_0'. use the below option to re-target to 'psu_cortexr5_0'. Doing this will clear the BSP and application setting"one on this boot domain

Re-target to psu_cortexr5_0

Modifying Source Code for FSBL

When boot component generation is selected in the platform generation phase, FSBL applications are created within the platform component. To modify the source code of these applications, follow the steps below.

1. To modify the source code for FSBL, go to the corresponding platform and expand the Source.
2. Expand the `zynqmp_fsbl_bsp` folder and modify the source files inside.
3. Save your changes and rebuild the platform with the new changes.

Note: To reset domain/BSP sources anytime, click the **Regenerate BSP** option on the Board Support Package overview page.

Note: An alternative way to update the FSBL and PMUFW source code is to follow the instructions in [Modifying the Domain Sources \(Driver and Library Code\)](#).

Modifying the Domain Sources (Driver and Library Code)

To add/modify the domain sources (driver and library code) using the AMD Vitis™ software platform, you must create your own repository with all the required files including the `.mld/.mdd` files and the source files. The installed driver and library code are located in the `<Vitis_Install_Dir>/data/embeddedsw` directory. A driver or library code component includes source files in the `src` directory and metadata in `data` directory. In the `.mld/.mdd` file, bump up the driver/library version number and add this repository to the Vitis software platform.

The Vitis software platform automatically infers all the components contained within the repository and makes them available for use in its environment. To make any modifications, you must make the required changes in the repository. Building the application gives you the modified changes.

Creating a Software Repository

A software repository is a directory where you can install third-party software components as well as custom copies of drivers, libraries, and operating systems. When you add a software repository, the AMD Vitis™ software platform automatically infers all the components contained within the repository and makes them available for use in its environment.

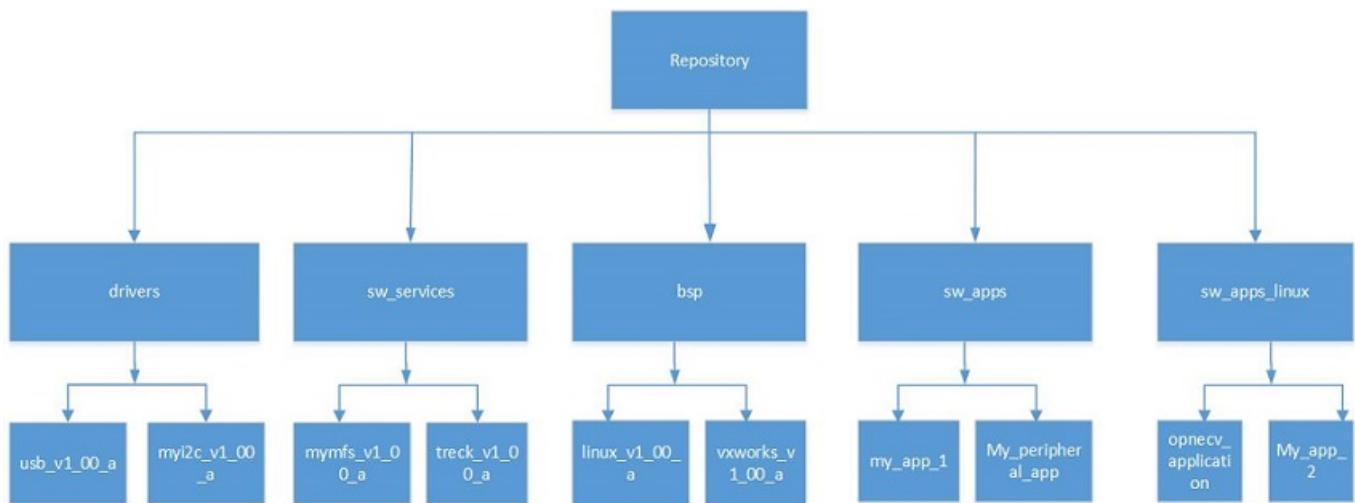
Your Vitis software platform workspace can point to multiple software repositories. The scope of the software repository can be global (available across all workspaces) or local (available only to the current workspace). Components found in any local software repositories added to a Vitis software platform workspace take precedence over identical components, if any, found in the global software repositories, which in turn take higher precedence over identical components found in the Vitis software platform installation.

A repository in the Vitis software platform requires a specific organization of the components. Software components in your repository must belong to one of the following directories:

- drivers: Used to hold device drivers.
- sw_services: Used to hold libraries.
- bsp: Used to hold software platforms and board support packages.
- sw_apps: Used to hold software standalone applications.
- sw_apps_linux: Used to hold Linux applications.

Within each directory, sub-directories containing individual software components must be present. The following diagram shows the repository structure.

Figure 26: Repository Structure



Adding the Software Repository

1. Select **Vitis** → **Embedded SW Repositories**.
2. To add the repository you created in [Creating a Software Repository](#), follow one of these two steps:
 - To ensure that your repository driver/library repository is limited to the current workspace, click **+** to add it under **Local Repositories**.
 - To ensure that your repository driver/library repository is available across all workspaces, click **+** to add it under **Global Repositories**.
3. Click **OK** to add the repository.

Resetting BSP Sources for a Domain

Execute the following steps to reset the source files of a domain's BSP.:

1. Select your platform component, expand Settings, click the `vitis-comp.json` file and select the appropriate domain.
2. Click **Regenerate BSP**. This resets the sources for the domain/BSP selected.

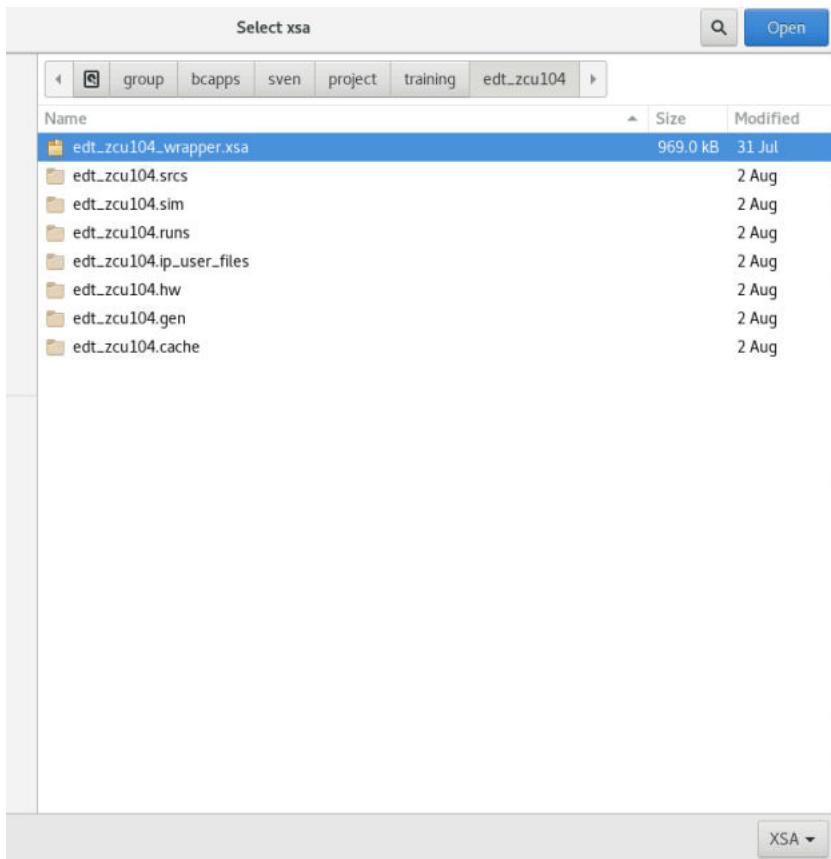
Note: Only the source files are reverted back to their original state. The settings however, are retained.

Updating the Hardware Specification

The AMD Vitis™ software platform allows you to update a platform project with a new hardware by updating the software components under the hood. If your AMD Vivado™ project and its exported XSA are updated, this workflow needs to be executed manually so that the Vitis software platform can get the updated hardware specification. You can edit the settings after the software platform adjusts the software components as per the new hardware.

To change the hardware specification file of the platform project, follow these steps:

1. Right-click the platform component in the component view, and expand Settings, open the `vitis-comp.json`.
2. Click **Update XSA**.
3. Specify the source hardware specification file in pop-up window.



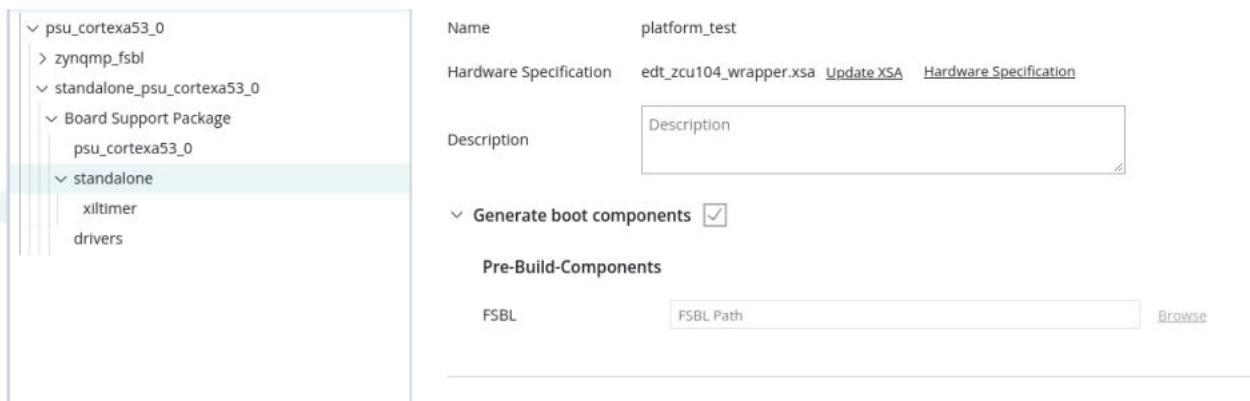
4. Click **open** to select the XSA file.

Reading Hardware Specification

The AMD Vitis™ software platform allows you to read the hardware platform information under the hood.

Execute the following steps to read the hardware information.

1. Right-click the platform component in the component view, expand **Settings**, and open the **vitis-comp.json** file.
2. To access the hardware information about your hardware platform, click **Hardware Specification**. For example, processor address information and PL IP address information.



Note: Clicking on the **Hardware Specification** has the same effect as double-clicking on the XSA file.

Applications

In Vitis, an embedded application refers to a specific type of software application that is designed to run on embedded systems or embedded platforms. Embedded applications are typically characterized by the following traits within the context of Vitis:

- **Targeted for Embedded Systems:** Embedded applications are intended to run on embedded hardware systems, which are typically resource-constrained and designed for specific tasks. These systems can range from small microcontrollers to more complex devices like embedded FPGAs or SoCs (System on Chips).
- **Real-Time or Resource-Constrained:** Embedded applications often need to meet real-time constraints or operate within tight resource limitations. They must efficiently use available CPU, memory, and I/O resources to perform their tasks reliably and predictably.
- **Diverse Use Cases:** Embedded applications in Vitis can serve a wide range of purposes, from controlling IoT (Internet of Things) devices, managing sensors and actuators, running real-time control algorithms, to performing signal processing, communication, and more.
- **Hardware Integration:** Embedded applications might interact closely with hardware components and peripherals. They often require specific device drivers and low-level hardware access to interface with sensors, motors, communication interfaces, and other embedded hardware.
- **Development Environment:** Vitis provides a development environment that allows developers to create, compile, and deploy embedded applications on their target hardware platform. It includes tools for code development, debugging, and performance optimization.
- **Cross-Compilation:** Embedded applications are often cross-compiled, meaning they are developed on a host computer but compiled to run on the target embedded system with a different architecture.

Creating application component

Creating an Application Component from Examples

You can create an application component from the example template by following steps:

1. Go to **View → Examples**, or click **File → Component From Example**, or click the tool bar on the left side or use **Ctrl + Shift + R** shortcut keys to open the example view.
2. Click the example in the example list.
3. Click the **Show details** to the requirement of the template.
4. Click **Create Application Component from Template**. You can refer to [Creating an Application Component](#) chapter for the following steps.

Creating an Application Component

Applications are linked to the standalone domain generated BSP or runs on the Linux domain operating system. A platform is required before creating an application.

To create an Application component you can select **File → New Component → Application**.



TIP: Alternatively you can select **Create Embedded Application** from the **Embedded Development** group of the **Welcome** page.

1. Input the Application component name and specify the location. Click **Next**
2. Select a fixed platform and select **Next**.

Note: If you do not have an existing fixed platform, refer to chapter: [Creating a Platform Component from XSA](#) to create a platform. If your target platform is not in the list, you can add it by clicking the '+' button.

3. Select the processor domain to run your application in, and select **Next**.

Note: If the domain is not suitable for your application, click **+ create new** to add a new domain and specify the operation system and processor according to your requirement and click **Next**.

4. Review the summary page of the Application component and select **Finish** to create the component.
5. Import source files

With the Application component created for the specific processor domain of a fixed platform and the source files imported, you can build or configure your building settings.

Creating a System Project Component

A system project can be created by following steps:

1. Go to **File→New Component→System Project**. The New System Project dialog is automatically displayed.

Note: You can create a system project from the Welcome page with the Create System Project shortcut.

2. Input the System project name and System project location. Click **Next**.
3. Select the Platform for System project. Click **Next**.

Note: If your platform is not in the Repo list, you can click the + button to add your platform in another workspace. Alternatively, you can refer to [Creating a Platform Component from XSA](#) to create a new platform and add it to the Repo list.

4. In the Embedded Component Path setup dialog, there is no need to set the Image path when performing embedded development. Click **Next**.

Note: The Embedded Component Path setting is used for acceleration application development. During acceleration development, you need to specify the rootfs, kernel image and boot components to generate final `sd_card.img` file.

5. Review the summary of the system project information and click **Finish**. The system project is displayed in the component view.

Managing Multiple Applications in a System Project Component

A system project can contain multiple applications that can run on a device simultaneously. Two applications for the same processor cannot be held together in one system project. For example, on an AMD Zynq™ UltraScale+™ MPSoC device, a standalone application running on a Cortex®-A53 and an application on a Cortex®-R5F can be held in the same system project if they are expected to run at the same time. However, an application on a Cortex-A53 and an application in Linux *cannot* be combined in one system project, because these applications use the same Cortex-A53 processors.

The following steps illustrate the flow to add two applications to one system project:

1. Click the system project and expand **Settings**, click **vitis-sys.json** to open the project.

Note: `Vitis-sys.json` is the system project setting file for configuring the system project.

2. Click **System Project Settings→Components→Adding Existing Components**.
3. Click the Application popup view and select the application components in your workspace.

Building Projects

The first step in developing a software application is to create a board support package to be used by the application. Then, you can create an application project.

When you build an executable for this application, Vitis automatically performs the following actions. Configuration options can also be provided for these steps.

1. The Vitis software platform builds the board support package. This is sometimes called a platform.
2. The Vitis software platform compiles the application software using a platform-specific `gcc/g++` compiler.
3. The object files from the application and the board support package are linked together to form the final executable. This step is performed by a linker which takes as input a set of object files and a linker script that specifies where object files should be placed in memory.

The following sections provide an overview of concepts involved in building applications.

Build Configuration Settings

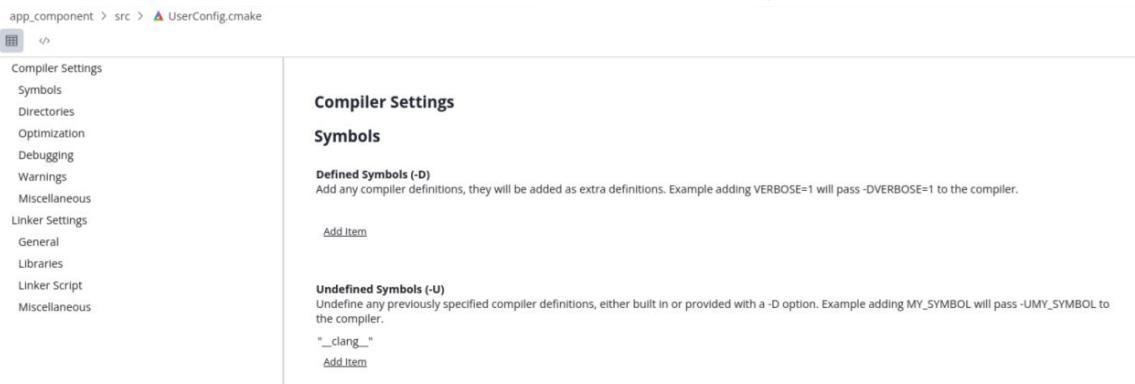
You could modify the build configurations manually. Each build configuration can customize:

- Compiler settings: debug and optimization levels
- Macros passed for compilation
- Linker settings

Adding Symbols or Definitions

Definitions and symbols are tokenized and processed as if they have appeared during a preprocessor translation phase in a `#define` directive. You can add or remove symbols in the Vitis IDE with the following steps:

1. Click your application component in the Component view and expand **Settings**. Open the `UserConfig.cmake` file under settings directory.
2. Under Compiler Settings, select **Symbols**.
3. Click the **Add Item** (⊕) button to add defined or undefined symbols.



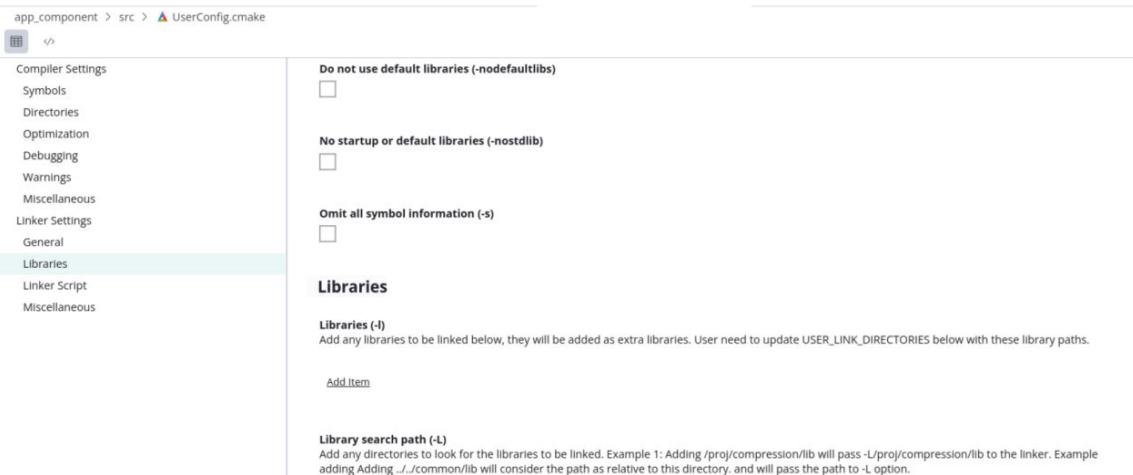
Note: `UserConfig.cmake` supports GUI format and text format. You can click these icons </> to access the source editor to view the source code and modify the settings in text format.

Adding Libraries and Library Paths

You can add libraries and library paths for Application projects. If you have a custom library to link against, you should specify the library path and the library name to the linker.

To set properties for your Application project:

1. Click your application component in Component view and expand **Settings**. Under settings directory open the `UserConfig.cmake` file.
2. Under **Compiler Settings**, select **Libraries**
3. In Libraries section, click **Add Item** to add the library name or the library path.



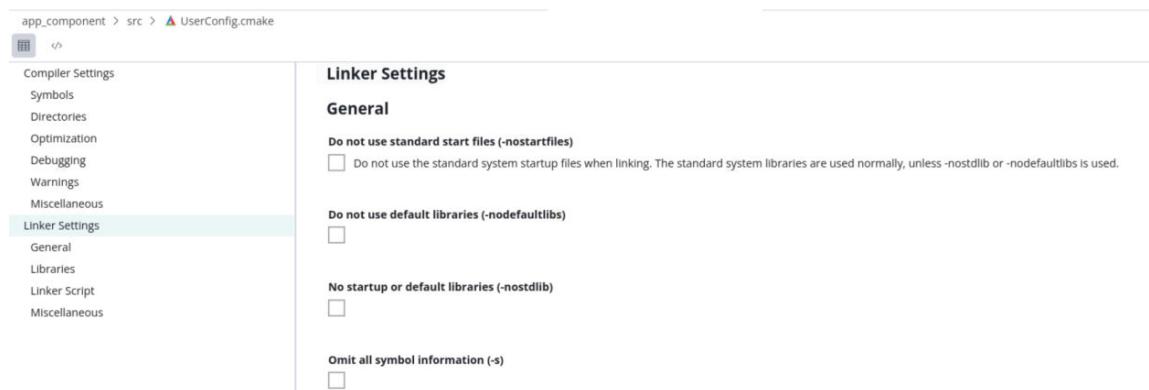
Note: `UserConfig.cmake` supports GUI format and text format. You can click the `</>` icons to access the source editor to view the source code and modify the settings in text format.

Specifying the Linker Options

You can specify the linker options for Application components. Any other linker flags not covered in the Tool Settings can be specified here.

To set properties for your application component:

1. Click your application component in Component view and expand **Settings**. Open the `UserConfig.cmake` file under settings directory.
2. Under **Compiler Settings**, select **Linker Settings**.
3. Click the check-box to enable or disable the Linker flags.



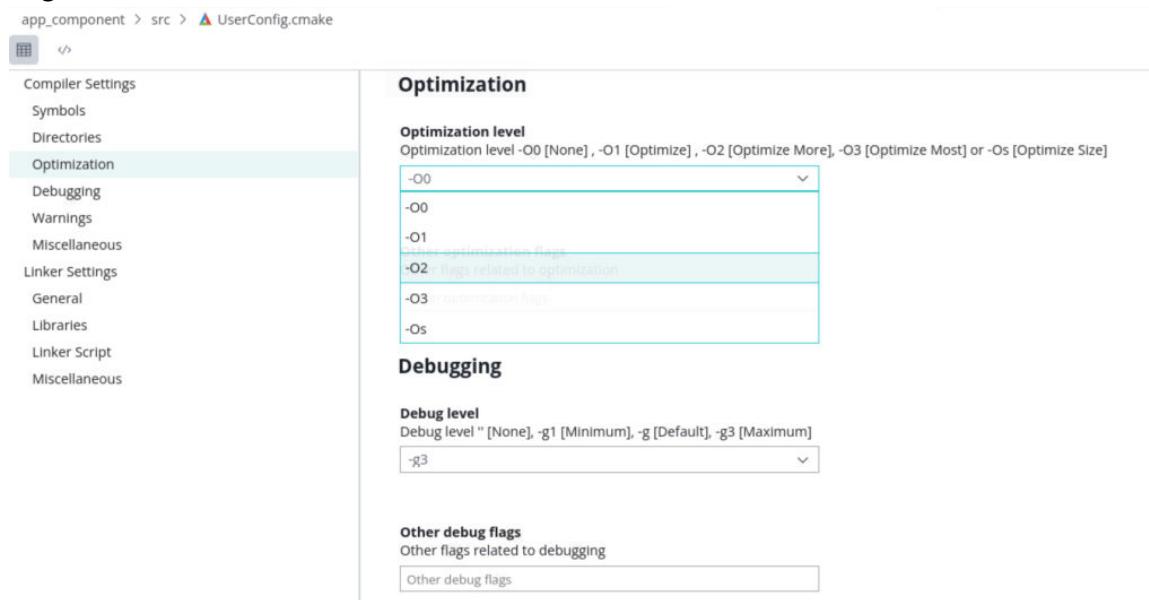
Note: UserConfig.cmake supports GUI format and text format. You can click the </> icons to access the source editor to view the source code and modify the settings in text format.

Specifying Debug and Optimization Compiler Flags

The Vitis software platform assigns a default optimization level and debug flags for the application component. You can change the default value for your application component.

To set properties for your project:

1. Click your application component in component view and expand **Settings**. Open the `UserConfig.cmake` file under settings directory.
2. Under Compiler Settings, select **Optimization**.
3. In the Optimization section, select the optimization level by clicking the drop-down button. Debugging section is under the Optimization section. Similarly, click the drop-down button to change the debug level. You can input other optimization flags or debug flags in the other flags field.



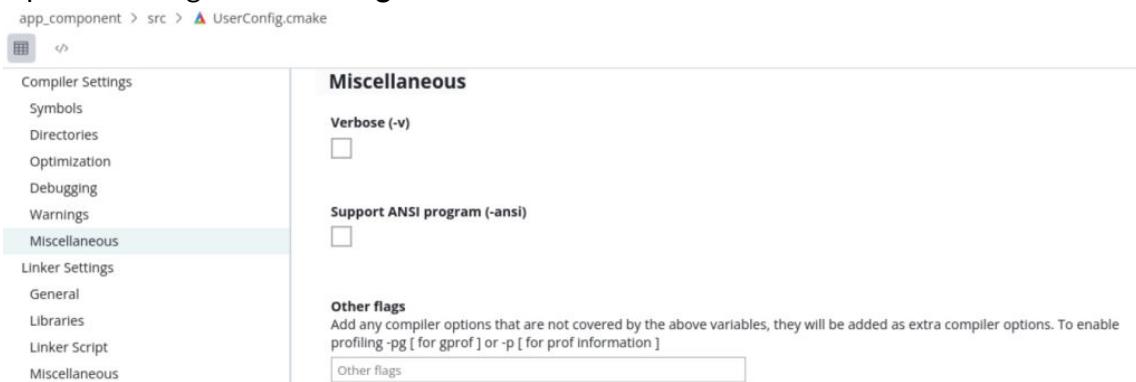
Note: UserConfig.cmake supports GUI format and text format. You can click the </> icons to access the source editor to view the source code and modify the settings in text format.

Specifying Miscellaneous Compiler Flags

You can specify any other compiler flags in the Miscellaneous section.

To set properties for your project:

1. Click your application component in component view and expand **Settings**. Open the UserConfig.cmake file under settings directory.
2. Under Compiler Settings, select **Miscellaneous**.
3. You can enable **Verbose** or **Support ANSI** program support by clicking the check-box. You can input other flags in **Other Flags** field.



Note: UserConfig.cmake supports GUI format and text format. You can click the </> icons to access the source editor to view the source code and modify the settings in text format.

Linker Scripts

The application executable building process can be divided into compiling and linking. Linking is performed by a linker that accepts linker command language files called linker scripts. The primary purpose of a linker script is to describe the memory layout of the target machine, and specify where each section of the program should be placed in memory.

Note: Only standalone applications need linker script. Linux OS helps managing the memory allocation, and thus it does not need a linker script.

The Vitis software platform provides a linker script generator to simplify the task of creating a linker script for GCC. The linker script generator in the Vitis IDE examines the memory nodes in the System Device Tree (SDT), to determine the available memory sections.

Note:

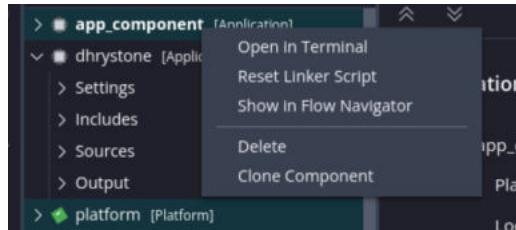
- For multiprocessor systems, each processor runs a different ELF file, and each ELF file requires its own linker script. Ensure that the two ELF files do not overlap in memory.

- The default linker always points to the DDR memory address available in memory. If you are creating an application under a given hardware/domain project, the memory overlaps for the applications.

Generating a Linker Script for an Application

Execute the following steps to generate a linker script for an application component or delete the linker script created with the application component.

1. Right click the **app_component** in the component view.
2. Select **Reset Linker Script** in the pop-up window.



3. Click **OK**.

If there are errors, they must be corrected before you can build your application with the new linker script.

Note: Select the appropriate option when a message view appears, to overwrite the file whether the linker script exists or not. Click **OK** to overwrite the file or **Cancel** to cancel the overwrite.

Manually Adding the Linker Script

If you want to manually add the linker script for your application project, do the following:

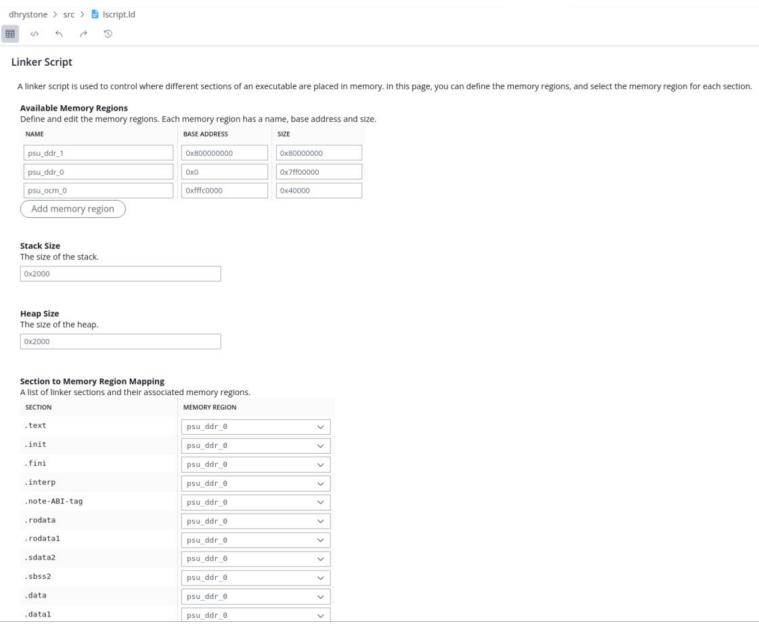
1. Select the application component in the Component View and then expand Settings. Right click **UserConfig.cmake** to open it.
2. Right click **Linker Script** to go to linker script setting section.
3. Click on **Browse** to select your own linker script file.

Modifying a Linker Script

There are multiple ways to update the linker script.

Linker script is located with the source code of your application component. Update it by executing the following steps.

1. Select the application component in the component view and expand Sources. Right click **lscript.ld** to open it.



2. You can view the source code of the linker file by clicking the `</>` button.
3. Undo/Redo: undo or redo the last actions for linker file.
4. You can reset the linker script by clicking the **Reset linker script** button..

The linker script editor provides the following functionality.

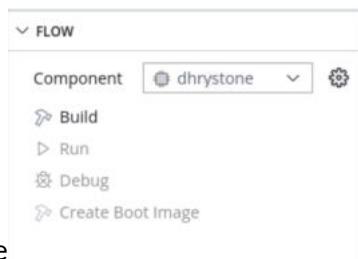
Note: This linker script supports text format as well. You can click `</>` button to change to text format to do modification.

Table 5: Linker Script Editor Functionality

Name	Function
Add Memory Regions	This section lists the memory regions specified in the linker script. You can add a new region by clicking on the Add button to the right. You can modify the name, base address and size of each defined memory region.
Stack and Heap Sizes	This section displays the sizes of the stack and heap sections. Simply edit the value in the text box to update the sizes for these sections.
Section to Memory Region Mapping	This section provides a way to change the assigned memory region for any section defined in the linker script. To change the assigned memory region, simply click on the memory region to bring a drop-down menu from which an alternative memory region can be selected. You can select several or whole section and click drop-down button to change to a memory region together.

Building the Application Component

After setting the build configuration and modifying the linker, you can start to build the application.



1. Go to flow navigate
2. Select the application component by clicking the drop-down button in the component field.
Note: Component is automatically selected in the flow navigator with the component in the component view.
3. Click **Build** to build your application component.

Once the build is complete, you can check the output in the output directory under the application component in component view.

Reading ELF Disassembly

This task is for you to view the disassembling code.

After application compilation is completed, you can get the application ELF file in the output directory. You can view the disassemble view of the code by double clicking the **ELF file**.

Changing a Referenced Domain

Note: This function is not supported with the Unified Vitis IDE. Change to the classic Vitis IDE and refer to 2023.1 documentation, if you wish to access this function.

Creating a Library Project

This feature is not yet supported by AMD Vitis™ Unified IDE. Use Classic AMD Vitis™ IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Creating a User Application Template

This feature is not yet supported by AMD Vitis™ Unified IDE. Use Classic AMD Vitis™ IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Using Custom Libraries in Application Projects

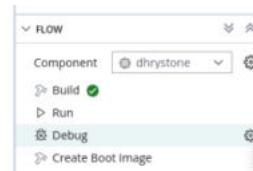
This feature is not yet supported by AMD Vitis™ Unified IDE. Use Classic AMD Vitis™ IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Run, Debug, and Optimize

Launch Configurations

To debug, run, and profile an application, you must create a launch configuration that captures the settings for executing the application. To do this, go to the Flow Navigator and select the application component, right-click on the **Open Settings** next to Run or Debug.

Figure 27: Flow Window



Note: The Open Settings command is a hidden icon. Hover the cursor over Flow Navigator next to either Run or Debug to view it.

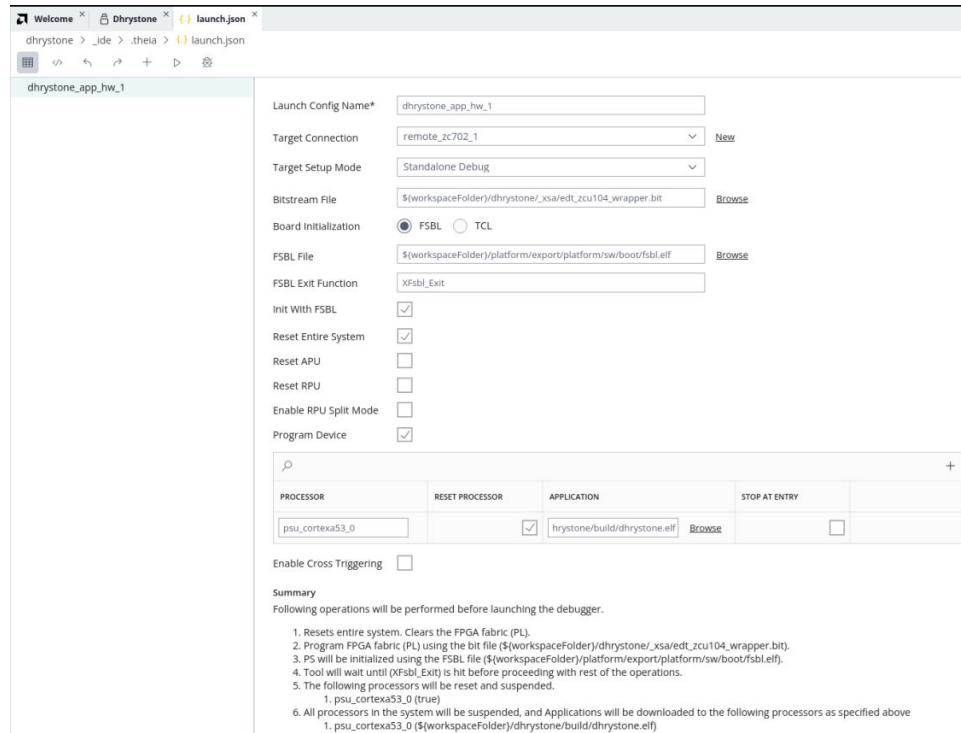
The `launch.json` file is opened to let you edit the launch configurations. The Launch Configuration editor has a toolbar menu at the top, as shown in the following figure. The specific contents of the launch configuration vary depending on the component or the project selected.

Launch Configurations

The commands in the Launch Configuration toolbar include:

- **Settings Form / Source Editor:** You can toggle between the GUI view and Text Editor view of the `launch.json` file with these commands.
- **Undo/Redo:** To undo or redo the last actions.
- **Add Configuration:** You can add a configuration by clicking the **+** button. The default launch configurations for System projects are Emulation SW, Emulation HW, and Hardware.
- **Run / Debug:** To launch run or debug using the currently selected launch configuration.

Figure 28: Launch Configurations



Hover your cursor over the configuration name and the Duplicate and Delete commands appear. Select **Delete** to remove the launch configuration, or **Duplicate** to copy the launch configuration.

In each configuration, you can update the settings to configure the tool prior to running or debugging the component or project. After setting up the launch configuration, you can select Run or Debug commands in the Launch Configuration editor, or by selecting Run or Debug from the Flow Navigator and specifying a launch configuration if more than one is available.

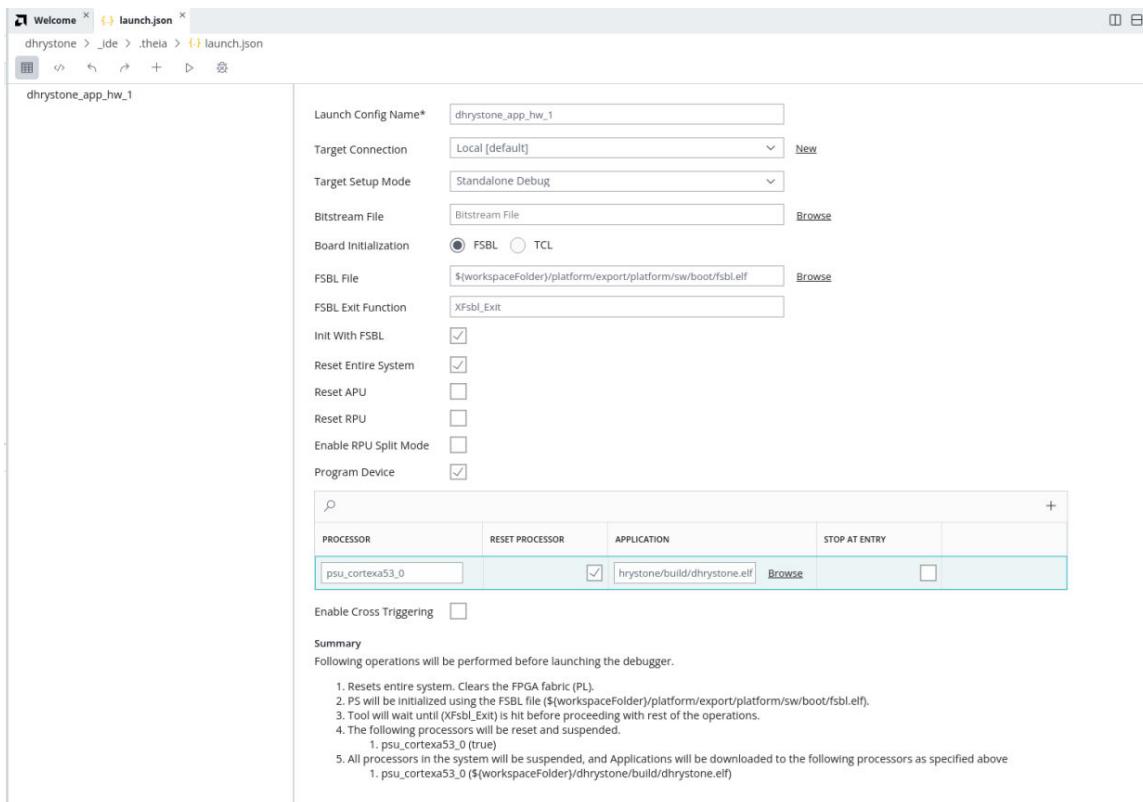
Main Page

The main view has the following options:

- **Launch Config Name:** You can specify the configuration name
- **Target Connection:** In the connection field, you can select a target or create a target connection by clicking **New** in the connection field.
- **Target Setup Mode:** You can select a standalone debug or attach a running target.
- **Bitstream File:** You can specify the bit file by browsing to corresponding directory
- **Board Initialization:** You can use the FSBL to initialize the board or use Tcl to initialize the board.
- **FSBL file:** Specify the FSBL file to do initialization

- **FSBL Exit Function:** Specify the FSBL exit function
- **Init With FSBL:** Use FSBL to initialize the PS.
- **Reset Entire System:** Perform a system reset if there is only one processor in the system
- **Reset APU:** Reset all the APU cores
- **Reset RPU:** Reset all the RPU cores
- **Enable RPU Split Mode:** Put RPU cores in split mode so that they can be used independent of each other
- **Program Device:** Allow tool to program the bit file to the device
- **Enable Cross Triggering:** Enable the cross trigger function

Figure 29: Launch Configuration Window

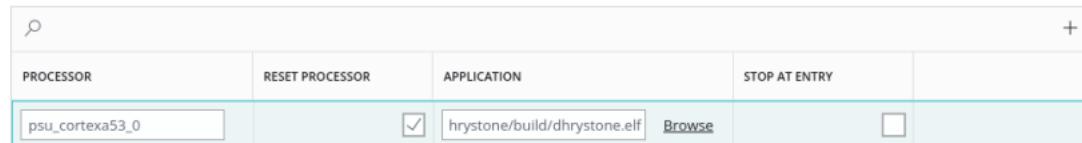


Note: The summary of this launch configuration file is displayed at the bottom.

Application Part

In the Launch configuration main view, you can set up the details for your application project and select the ELF file.

Figure 30: Debug Confrontation Application Part



- **PROCESSOR:** Specify the detailed processor
- **Reset Processor:** You can choose to reset the entire hardware system or the specific processor, or choose not to reset. Performing a reset ensures that there are no side effects from a previous debug session.
- **APPLICATION:** Specify the application file
- **STOP AT ENTRY:** Application stop at entry point.
- **+**: If you have multiple applications with different processors you can click + to add new processor and application configurations.

Note: Place your mouse over the configuration under the + column and the Edit and Delete commands appear. Select **Delete** to remove the one set of the application configuration, or **Edit** to edit the configuration..

Target Connections

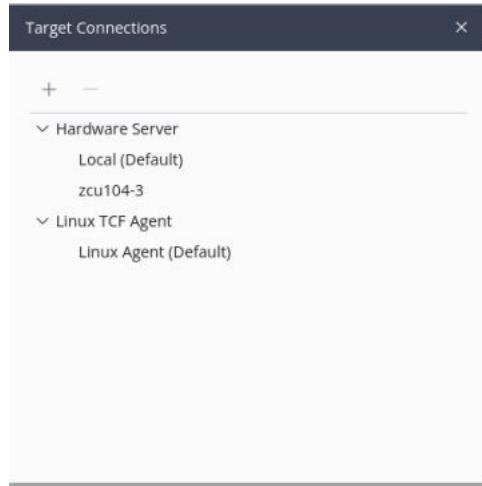
The Target Connections dialog box (🌐) allows you to configure multiple remote targets. It shows connected targets and gives you an option to add or delete target connections.

The Vitis software platform establishes target connections through the AMD `hw_server`. In order to connect to remote targets, the hardware server agent must be running on the remote host, to which the target hardware is connected.

The target connection has been extended to all utilities within the Vitis Unified IDE that deal with targets at runtime.

From the menu, select **Vitis** → **Target Connections** to open the target manager.

Figure 31: Target Connections



Creating a New Target Connection

You can configure the remote target details by adding a new connection in the Target Connections view.

To create new target connection:, do the following.

1. Right click the connection **Hardware Server** or Linux TCF Agent you want to add.
2. Click the **Add Target Connection** button (+) on the toolbar.
3. The Target Connection Details page opens.
4. In the Target Name field, type a name for the new remote connection.
5. Check the Set as default target check-box to set this target as default. The Vitis Unified IDE uses the default target for all the future interactions with the board.
6. In the Host field, type the name or IP address of the remote host machine. This is the machine that is connected to the target and the `hw_server` is running.
7. In the Port field, type the port number on which the `hw_server` is running. By default, the `hw_server` runs on port 3121.
8. Select **Use Symbol Server**, if the hardware server is running on a remote host.
9. Click **OK** to create a new target connection.

Note: Before clicking OK, you can click **Test Connection** to test the connectivity.

Setting Custom JTAG Frequency

You can operate at a different frequency supported by the JTAG cable by setting the JTAG frequency.

To set the JTAG frequency, execute the following steps.

1. Right click the connection **Hardware Server** or **Linux TCF Agent** you want to add.
2. Click the **Add Target Connection** button (+) on the toolbar.
3. The Target Connection Details page opens.
4. In the Target Name field, type a name for the new remote connection.
5. Check the Set as default target check-box to set this target as default. The Vitis Unified IDE uses the default target for all the future interactions with the board.
6. In the Host field, type the name or IP address of the remote host machine. This is the machine that is connected to the target and the `hw_server` is running.
7. In the Port field, type the port number on which the `hw_server` is running. By default, the `hw_server` runs on port 3121.
8. Select **Use Symbol Server**, if the hardware server is running on a remote host.
9. Click **Advanced** and select **Automatically discover devices ON JTAG chain** to view the JTAG device chain details.
10. From the Set custom frequency drop-down list, select the frequency.

Note: Current frequency can be the default frequency set by the server or the custom frequency set by a debug client.

11. Click **OK** to save the configuration and create a new target connection. The selected frequency is saved in the workspace and is used to set the frequency before executing a connect command for the selected device.
12. **Note:** If only one client is connected to the server, the frequency of the cable is reset to the default value whenever the connection is closed. However, in case of multiple clients connected to the server, it is not recommended to perform simultaneous debug operations from different clients.

Establishing a Target Connection

To establish a target connection, you can use either the local board or the remote board. By default, the local target connection is selected in the Target Connection view. You can confirm connections to the local board by checking the local connection.

To use a remote board to establish a target connection:

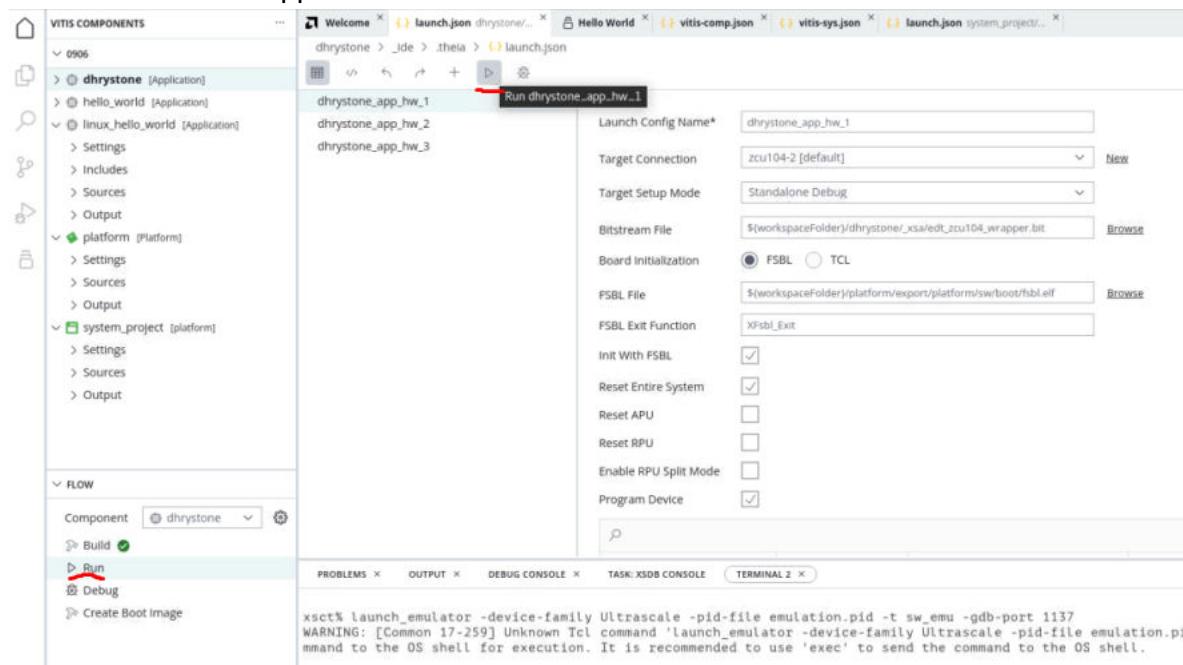
1. Ensure that the target is connected to the remote host.
2. Launch the `hw_server` manually on the remote host:
 - a. Take a shell on the remote host.
 - b. Source the setup scripts by using `C:/Xilinx/Vitis/<version>/settings64.bat` (or) `<Vitis_local_install_path>/ Vitis/<version>/settings64.csh`.
3. Run the `hw_server` on the machine that connects to the board.

Note: Ensure that the target (board) is connected to the remote host.

4. Select the port number and the hostname to create a target connection to the host running the hw_server.
5. Right-click the newly created target connection and select **Set As Default**.

Running the Application Component

1. Go to flow navigator and select the application component you want to run.
2. Click open settings to launch configuration. Refer to [Launch Configurations](#) to launch and configure the launch configuration file.
3. Refer to: [Creating a New Target Connection](#) for setting the target connection.
4. Click run to run the application.



Note: After running the application, you need to terminate the session by clicking the **Terminate** button in Debug view.

Debugging Application Component

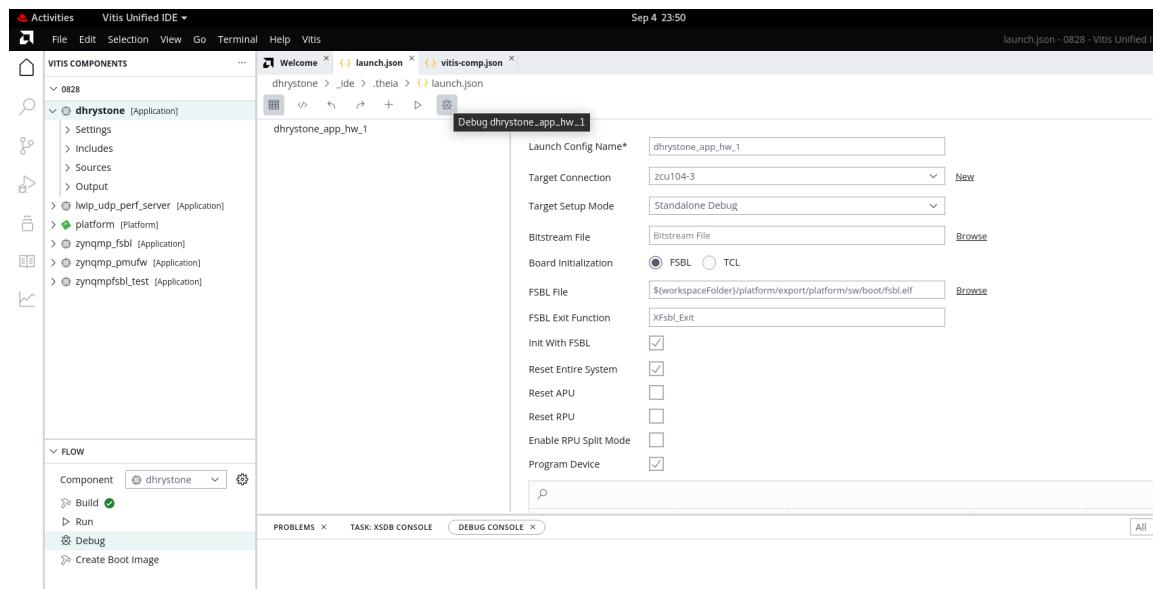
Starting Debug

The following chapters the steps to start the debug process.

Debugging Standalone Application Component

This topic describes how to use the Debugger to debug standalone applications.

1. Go to the Flow Navigator and select the application component.
2. Open**Debug Settings** and refer to [Launch Configurations](#) to create a new configuration for the application component.
3. Refer to: [Creating a New Target Connection](#) for setting the target connection.
4. Click debug to start debugging.



5. The debug view is displayed.

```

228 { One_Fifty     Int_1_Loc;
229   One_Fifty     Int_2_Loc;
230   One_Fifty     Int_3_Loc;
231   char          Ch_Index;
232   Enumeration   Enum_Loc;
233   Str_30        Str_1_Loc;
234   Str_30        Str_2_Loc;
235   int            Run_Index;
236   int            Number_Of_Runs;
237
238 /* Initializations */
239 init_platform();
240 xil_printf("Enter dmips prg\r\n");
241
242 Next_Ptr_Glob = &tmp_var1;
243 Ptr_Glob = &tmp_var2;
244
245 Ptr_Glob->Ptr_Comp = Next_Ptr_Glob;
246 Ptr_Glob->Discr  = Ident_1;
247 Ptr_Glob->variant.var_1.Enum_Comp = Ident_3;
248 Ptr_Glob->variant.var_1.Int_Comp = 40;
249 strcpy(Ptr_Glob->variant.var_1.Str_Comp,
250       "DHRYSTONE PROGRAM, SOME STRING");
251 strcpy(Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");
252
253 Arr_2_Glob[8][7] = 10;
254
255 /* Was missing in published program. Without this statement, */
256 /* Arr_2_Glob[8][7] would have an undefined value. */
257

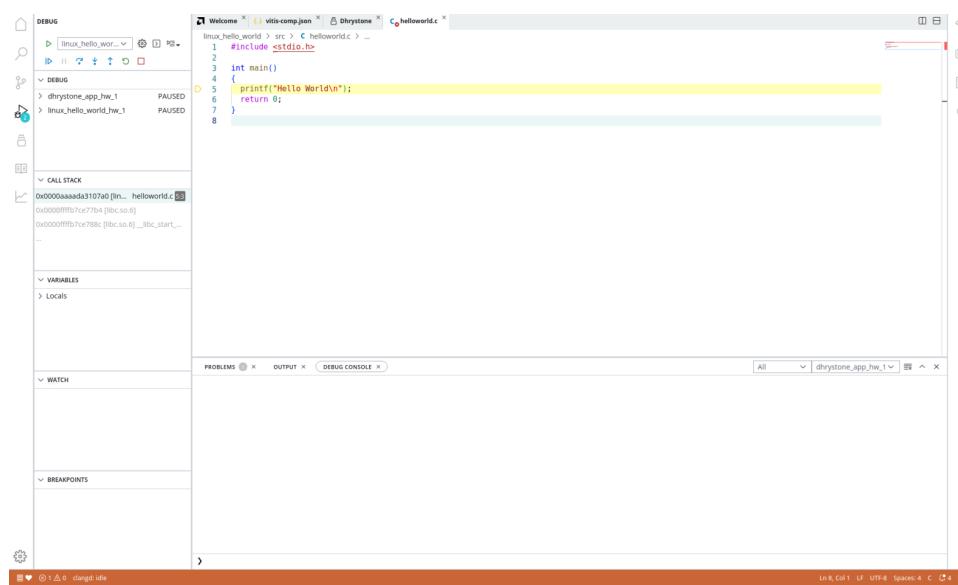
```

Debugging Linux Application Component

The following steps explains how to debug a Linux application component.

1. Go to the Flow Navigator and select the Linux application component you wish to debug.
2. Open **Debug Settings** and refer to [Launch Configurations](#) to launch and edit the configuration file.
3. Refer to: [Creating a New Target Connection](#) for setting the target connection.
4. Click **Debug** to start debugging.

The debug view is displayed.



Debugging Standalone Application Component on QEMU

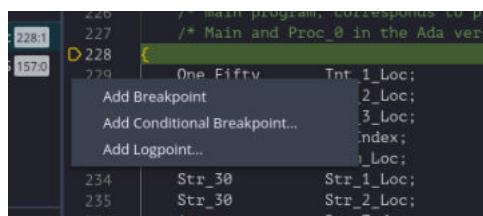
This feature is not yet supported by AMD Vitis™ Unified IDE. Use Classic AMD Vitis™ IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Setting Conditional Breakpoints

This feature assists you in setting conditional breakpoints during the debugging process

To add conditional breakpoints, execute the following steps.

1. Right click on the left margin before the line number.
2. Select **Add Conditional Breakpoint**.



3. Type any expression that evaluates to a Boolean and hit enter to add a conditional breakpoints.

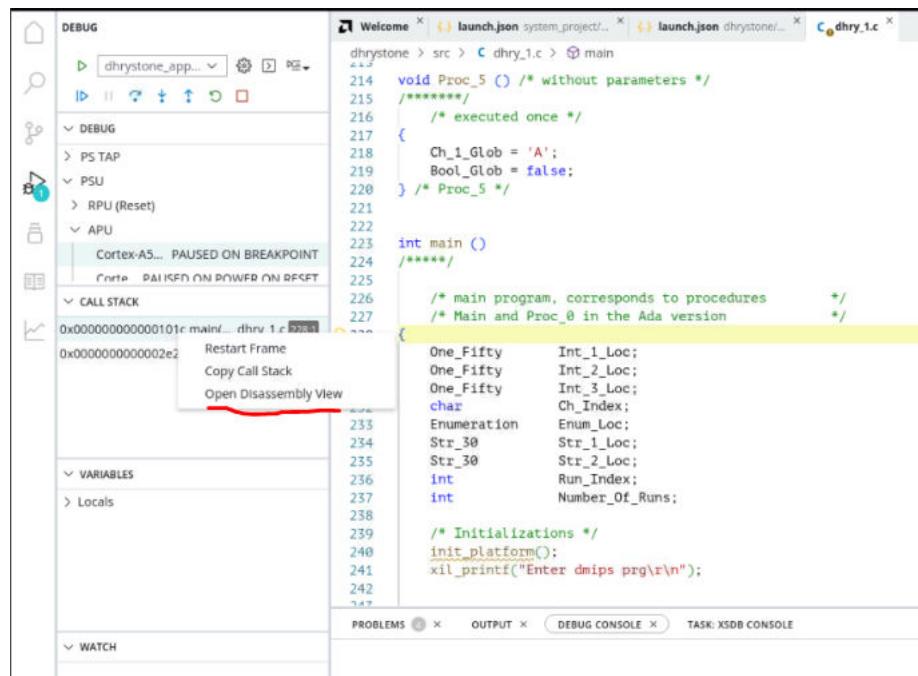
To change a normal breakpoint to conditional breakpoints, follow steps outlined below.

- Right click on an existing breakpoint
- Select **Edit Breakpoint**.
- Add the expression to change it to a conditional breakpoint.

Viewing Disassembly Code

To view the disassembly code, right click the function stack in the stack view after the debug session is started.

Figure 32: Viewing Disassembly Code



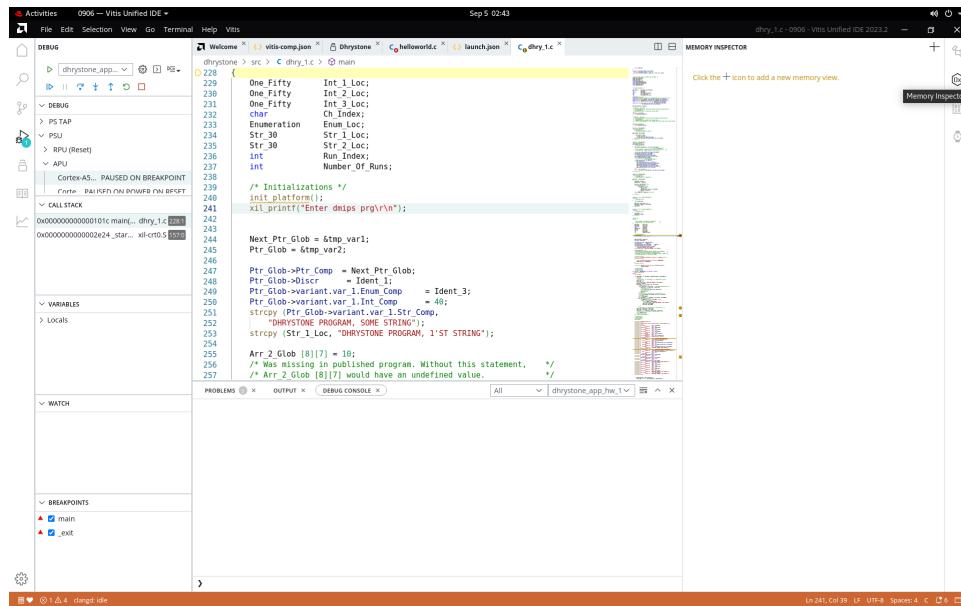
Next, select the Open Disassembly to view disassembly code is displayed.

Viewing Memory

Viewing the Value of a Certain Memory address

Follow the steps below for viewing the value of a certain memory address:

1. After the debug session has started, click **Memory Inspector** to view the memory information at the top right corner.



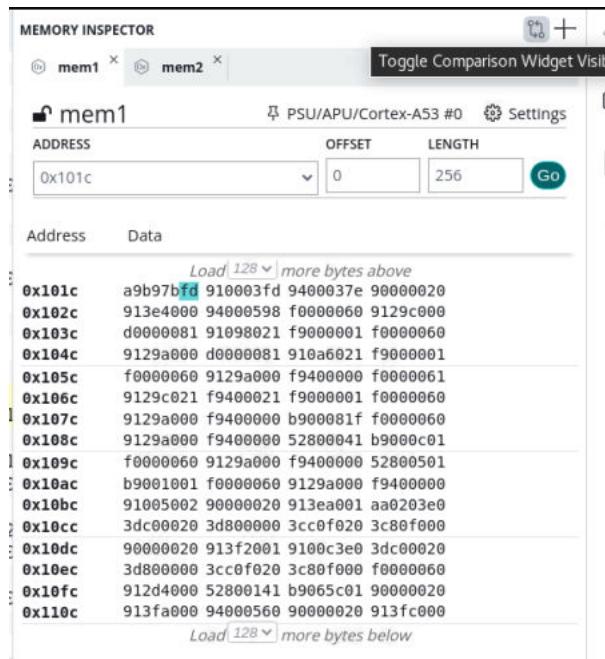
2. Click + icon to add a new memory view.
3. Set the memory address, offset, and length. Click Go.

The memory value is displayed in the Memory view.

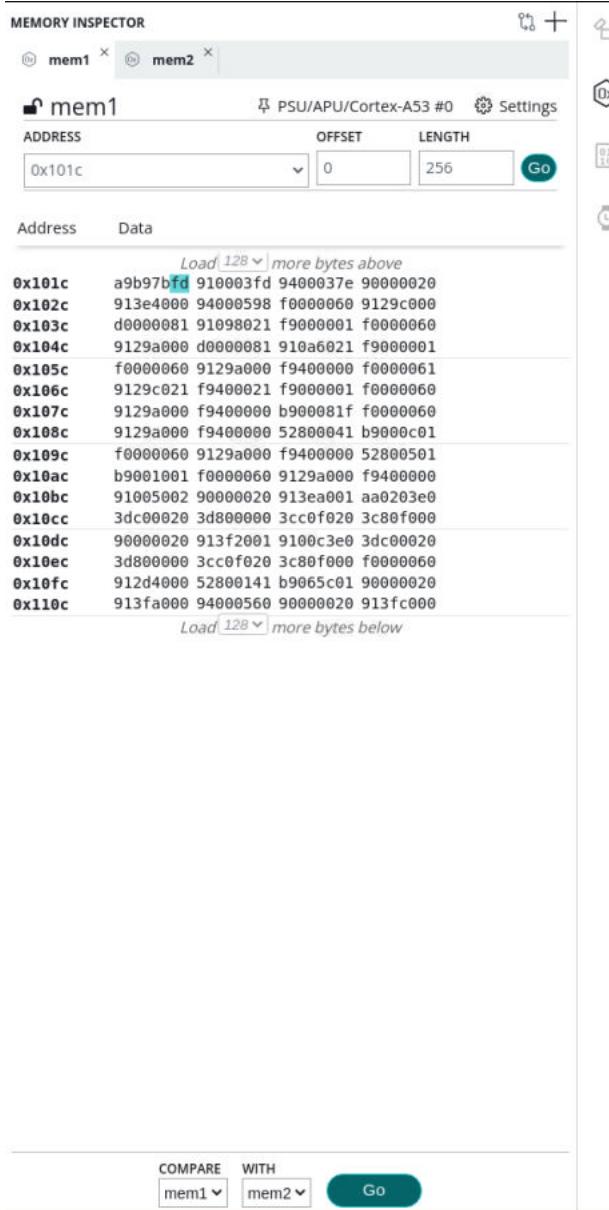
Comparing the Memory Value of Two Addresses

Follow the steps below to compare the value of the memory addresses:

1. Refer to [Viewing the Value of a Certain Memory address](#) to create two memory inspectors. Set the difference of the two inspectors.



2. Click **Toggle Comparison Widget Visibility** to compare the difference.
3. Select the memory you want to compare and click **Go**.



MEMORY INSPECTOR

mem1 x mem2 x

mem1 PSU/APU/Cortex-A53 #0 Settings

ADDRESS OFFSET LENGTH

0x101c 0 256 Go

Address Data

0x101c	a9b97b <code>fd</code> 910003fd 9400037e 90000020
0x102c	913e4000 94000598 f0000060 9129c000
0x103c	d0000081 91098021 f9000001 f0000060
0x104c	9129a000 d0000081 910a6021 f9000001
0x105c	f0000060 9129a000 f9400000 f0000061
0x106c	9129c021 f9400021 f9000001 f0000060
0x107c	9129a000 f9400000 b900081f f0000060
0x108c	9129a000 f9400000 52800041 b9000c01
0x109c	f0000060 9129a000 f9400000 52800501
0x10ac	b9001001 f0000060 9129a000 f9400000
0x10bc	91005002 90000020 913ea001 aa0203e0
0x10cc	3dc00020 3d800000 3cc0f020 3c80f000
0x10dc	90000020 913f2001 9100c3e0 3dc00020
0x10ec	3d800000 3cc0f020 3c80f000 f0000060
0x10fc	912d4000 52800141 b9065c01 90000020
0x110c	913fa000 94000560 90000020 913fc000

Load 128 more bytes above

Load 128 more bytes below

COMPARE WITH

mem1 mem2 Go

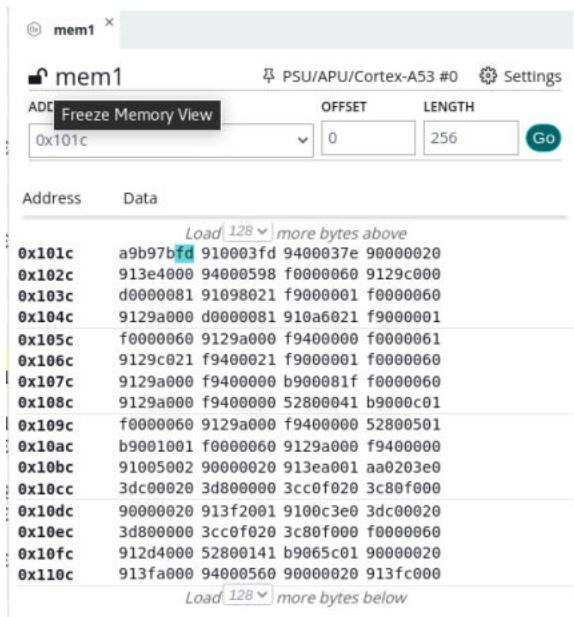
4. The comparison result is displayed in main view.

Freezing Memory Value

This feature keeps a snapshot of the memory contents until the tool is closed. You can compare the stored value or use the information to debug issues. To use this function, follow the steps below:

1. Refer to [Viewing the Value of a Certain Memory address](#) to create a new memory inspector.

2. Click the lock icon to freeze the memory view.



The screenshot shows the 'mem1' tab of the Memory View tool. The 'Freeze Memory View' button is highlighted in black. The table below shows memory addresses from 0x101c to 0x110c and their corresponding data values. The data values are mostly in hex format, with some being bolded (e.g., 0x101c, 0x102c, 0x103c, 0x104c, 0x105c, 0x106c, 0x107c, 0x108c, 0x109c, 0x10ac, 0x10bc, 0x10cc, 0x10dc, 0x10ec, 0x10fc, 0x110c) and some in decimal (e.g., 910003fd, 9400037e, 90000020, 913e4000, 94000598, f0000060, 9129c000, d0000081, 91098021, f9000001, f0000060, 9129a000, d0000081, 910a6021, f9000001, f0000060, 9129a000, 9129a000, f9400000, f0000061, 9129c021, f9400021, f9000001, f0000060, 9129a000, f9400000, b900081f, f0000060, 9129a000, f9400000, 52800041, b9000c01, f0000060, 9129a000, f9400000, 52800501, b9001001, f0000060, 9129a000, f9400000, 91005002, 90000020, 913ea001, aa0203e0, 3dc00020, 3d800000, 3cc0f020, 3c80f000, 90000020, 913f2001, 9100c3e0, 3dc00020, 3d800000, 3cc0f020, 3c80f000, f0000060, 912d4000, 52800141, b9065c01, 90000020, 913fc000, 913fa000, 94000560, 90000020, 913fc000). The table has 'Address' and 'Data' columns. There are also 'Load 128' buttons at the top and bottom of the data table.

Address	Data
0x101c	a9b97bf d 910003fd 9400037e 90000020
0x102c	913e4000 94000598 f0000060 9129c000
0x103c	d0000081 91098021 f9000001 f0000060
0x104c	9129a000 d0000081 910a6021 f9000001
0x105c	f0000060 9129a000 f9400000 f0000061
0x106c	9129c021 f9400021 f9000001 f0000060
0x107c	9129a000 f9400000 b900081f f0000060
0x108c	9129a000 f9400000 52800041 b9000c01
0x109c	f0000060 9129a000 f9400000 52800501
0x10ac	b9001001 f0000060 9129a000 f9400000
0x10bc	91005002 90000020 913ea001 aa0203e0
0x10cc	3dc00020 3d800000 3cc0f020 3c80f000
0x10dc	90000020 913f2001 9100c3e0 3dc00020
0x10ec	3d800000 3cc0f020 3c80f000 f0000060
0x10fc	912d4000 52800141 b9065c01 90000020
0x110c	913fa000 94000560 90000020 913fc000

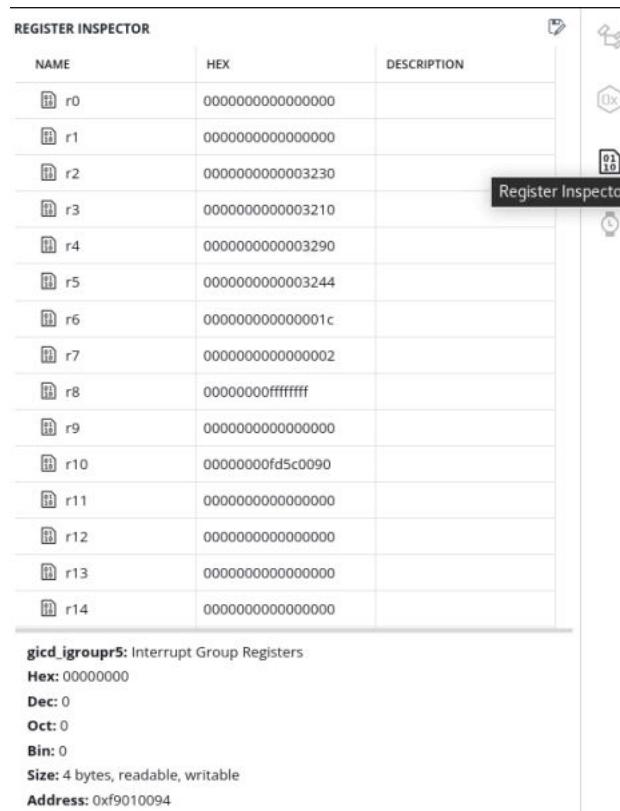
The value can be viewed until the tool is closed.

Viewing Registers

The Registers Inspector lists all registers, including general purpose registers, system registers and IP registers. For example, for AMD Zynq™ devices, the Registers view displays all the processor and co-processor registers when Cortex®-A9 targets are selected in the Debug view. The Registers view shows system registers and IOU registers when an APU target is selected.

To view the register's value, click the **Register Inspector** button after the debug view is opened. You can also open the Register Inspector from **View→Register Inspector**.

Figure 33: Register Inspector Window



The screenshot shows the Register Inspector window with a list of registers (r0 to r14) and a detailed view for gicd_igroupr5. The registers list has columns for NAME, HEX, and DESCRIPTION. The detailed view for gicd_igroupr5 shows its type as Interrupt Group Registers, hex value as 00000000, decimal value as 0, octal value as 0, binary value as 0, size as 4 bytes, and address as 0xf9010094.

NAME	HEX	DESCRIPTION
r0	0000000000000000	
r1	0000000000000000	
r2	0000000000003230	
r3	0000000000003210	
r4	0000000000003290	
r5	0000000000003244	
r6	0000000000000001c	
r7	0000000000000002	
r8	00000000fffffff	
r9	0000000000000000	
r10	00000000fd5c0090	
r11	0000000000000000	
r12	0000000000000000	
r13	0000000000000000	
r14	0000000000000000	

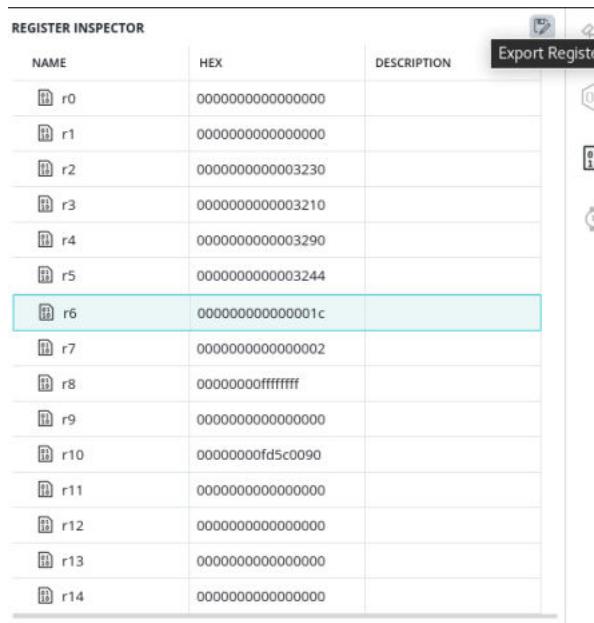
gicd_igroupr5: Interrupt Group Registers
Hex: 00000000
Dec: 0
Oct: 0
Bin: 0
Size: 4 bytes, readable, writable
Address: 0xf9010094

You can modify the editable field values during debug. You can also click the corresponding register name to view the detailed information.

Exporting Registers from the Vitis IDE

This feature allows you to export the registers present in a target processor to a text file and read all the register values more easily, which can be helpful while debugging.

1. Select the application component and start the debug session. See [Starting Debug](#) for more details.
2. Open the Register Inspector view. See [Viewing Registers](#) for details.
3. Click the **Export Registers** button to export the register.



NAME	HEX	DESCRIPTION
r0	0000000000000000	
r1	0000000000000000	
r2	0000000000003230	
r3	0000000000003210	
r4	0000000000003290	
r5	0000000000003244	
r6	000000000000001c	
r7	0000000000000002	
r8	00000000fffffff	
r9	0000000000000000	
r10	00000000fd5c0090	
r11	0000000000000000	
r12	0000000000000000	
r13	0000000000000000	
r14	0000000000000000	

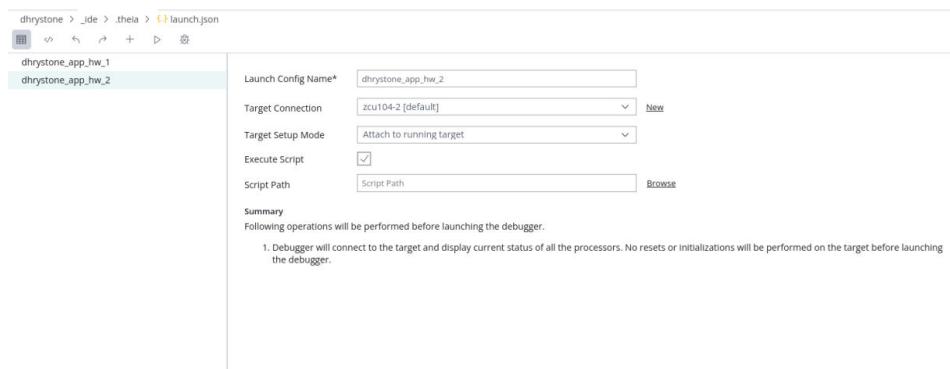
4. Add the following information on the next screen:
 - **Location:** Provide the location where you want to save the register dump file.
 - **Select registers/groups to export:** Select the list of registers to be dumped. You can uncheck any registers that are not required.
5. Click **OK** to dump the registers to the location you specified in the previous step.

Debugging an Application Component Already Running On a Target Device

The debugger allows debugging an application that is already running on the target device. For example, a standalone application component or Linux kernel booting from a flash device can be debugged using the debugger. The following are the steps for attaching to an application component running on the target.

Note: The debugger does not modify the state of the processors on the target device, but merely connects to them. You can halt the processors and debug from the current PC.

1. Create a target connection to the host to where the hardware board is connected. If the hardware board is connected to the same machine where the Vitis Unified IDE is running, this step can be skipped. In the subsequent steps that refer to remote board and remote connection, the default *Local connection* can be used.
 - a. See [Creating a New Target Connection](#) to create the target connection.
2. See [Launch Configurations](#) to launch a debug configuration. Set the **Target Setup Mode** as **Attach to running target** and select the target connection that was created.



3. Follow [Starting Debug](#) to start debug the application.

Set Up Path Mapping

Note: This function is not supported in the Vitis Unified IDE. Switch to classic Vitis IDE and refer to 2023.1 documentation, if you wish access this function.

Debugging an Application on the Emulator (QEMU)

Note: This function is not supported in the Vitis Unified IDE. Switch to classic Vitis IDE and refer to 2023.1 documentation if you wish access this function.

Running and Debugging Application Components under a System Project Together

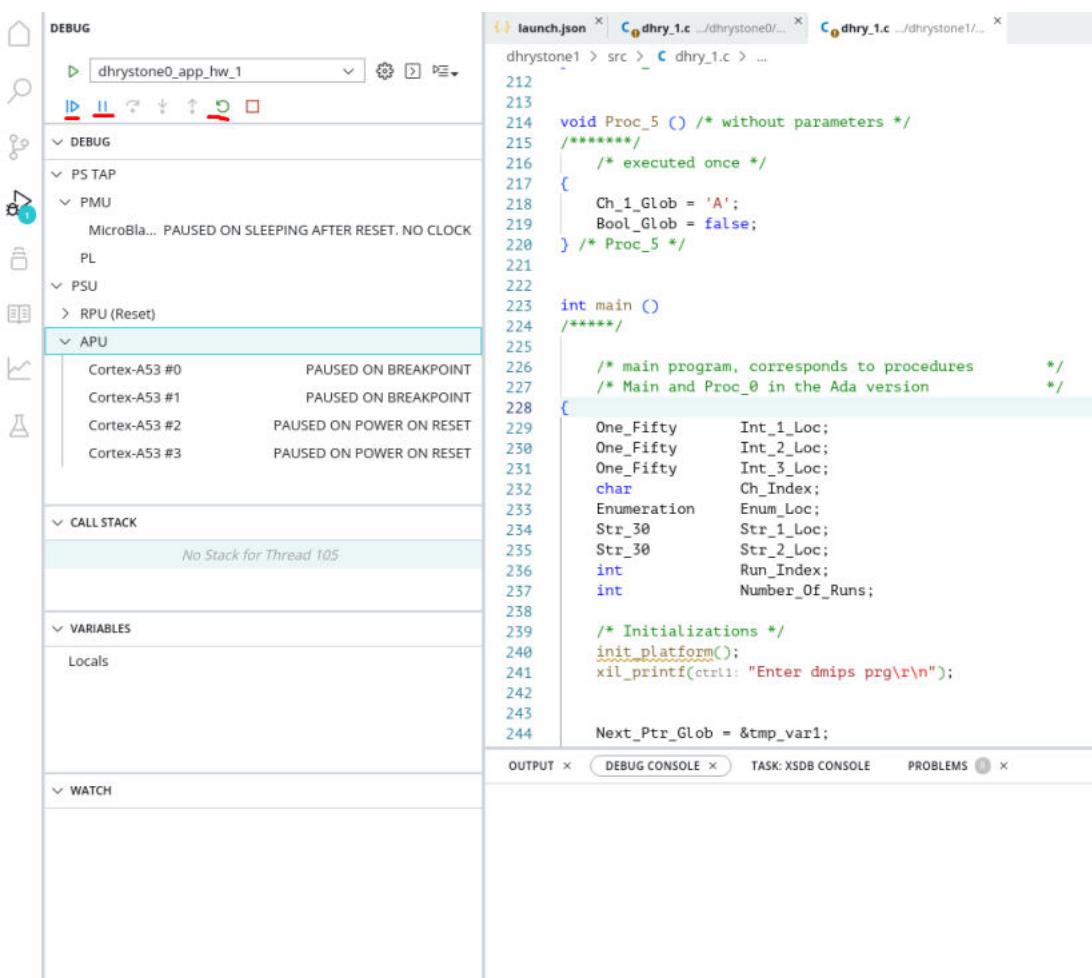
Each application component under a system project can run standalone. Application components under a system project can be launched together as well. The Vitis Unified IDE can download them one by one and launch them one after another. In debug mode, all applications stop at `main()`. The following steps detail the steps to run or debug application components under a system project together.

1. Select the system project in the Component View and go to flow navigator to start a run / debug launch configuration.

Note: Refer to [Launch Configurations](#) chapter to launch new configuration and refer to: [Creating a New Target Connection](#) for setting the target connection. Review the application, processor and target setting in the configuration page.

2. Click debug or run to start debugging or running system application. The debug session appears in the main view.

Note: Select the core against the corresponding application to do step over, step and step out or select the parent node to start, pause or resume all the cores.



Debugging on a Remote Board

The following describes the steps for debugging on a remote board.

1. Setting up the Remote System Environment:

- Run `hw_server` with a non-default port (for example: 3122) to enable remote connections. Use the following command to launch the `hw_server` on port 3122:

```
the hw_server -s TCP::3122
```

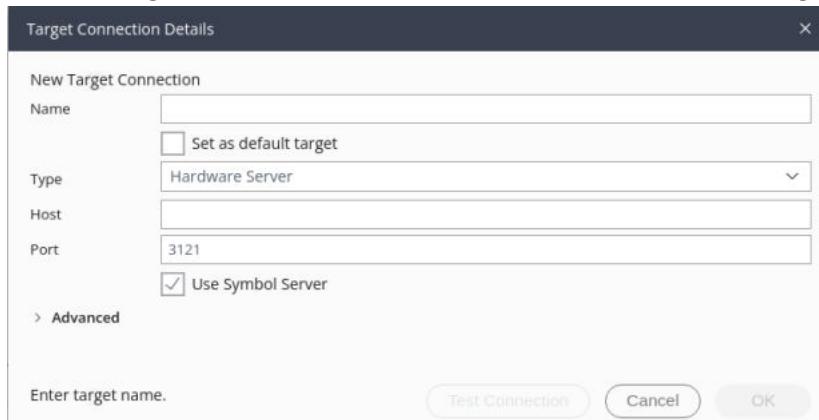
- Make sure your board is correctly connected.
- In a `cmd` window on the host machine, check the IP address:

```
Windows IP Configuration

Ethernet adapter Local Area Connection:

Connection-specific DNS Suffix . : xilinx.com xilinx.com
Link-local IPv6 Address . . . . : fe80::1562:9f81:b0c9:965z.11
IPv4 Address . . . . . : 172.23.19.161
```

2. Setup the Local System for Remote Debug:
 - a. Launch the Vitis Unified IDE.
 - b. Select the application component to debug.
 - c. Go to the Flow Navigator and click **Open Launch** configuration. See [Launch Configurations](#) for details.
 - d. For the Target Connection field, click **New** to create a new target connection.



- e. In the New Target Connection wizard, add the required details for the remote host that is connected to the target.
- f. In the **Target Name**, type a name for the target.
- g. In the **Host** field, enter the IP address or name of the host machine.
- h. In the **Port** field, enter the Port on which the hardware server was launched, for example 3122.
- i. Select **Use Symbol Server** to ensure that the source code view is available, during debugging the application remotely. Symbol server acts as a mediator between hardware server and the AMD Vitis™ Unified IDE.
- j. Click **OK**.
- k. Two available connections are displayed. In this case, `remote_zc702_1` is the remote connection.



- l. Click the **debug** button.

Note: For target connection creation, refer to: [Target Connections](#).

OS Aware Debugging

This feature is not yet supported by AMD Vitis Unified IDE. Use Classic Vitis IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Xen Aware Debugging

This feature is not yet supported by AMD Vitis Unified IDE. Use Classic Vitis IDE if this is a requirement for your project. Refer to 2023.1 documentation for details about how to use this feature in the classic IDE.

Debugging Self-Relocating Programs

Note: This function is not supported in the Vitis Unified IDE. Switch to classic Vitis IDE if you wish to access this option.

Debugging an Application Project Using the Emulator (Command-Line Flow)

Note: This function is not supported in the Vitis Unified IDE. Switch to classic Vitis IDE and refer to 2023.1 documentation, if you wish to access this option.

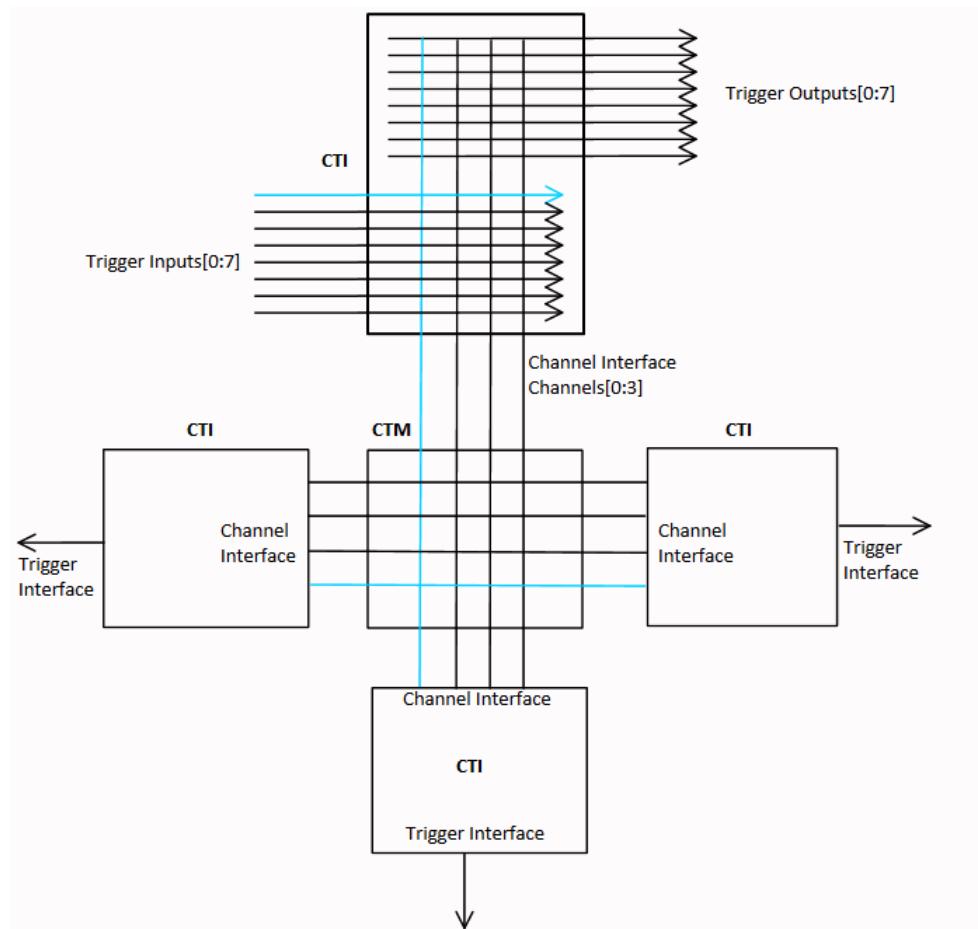
Cross-Triggering

Cross-triggering is supported by the embedded cross-triggering (ECT) module supplied by Arm. ECT provides a mechanism for multiple subsystems in an SoC to interact with each other by exchanging debug triggers. ECT consists of two modules:

- Cross Trigger Interface (CTI) - CTI combines and maps the trigger requests, and broadcasts them to all other interfaces on the ECT as channel events. When the CTI receives a channel event, it maps this onto a trigger output. This enables subsystems to cross trigger with each other.
- Cross Trigger Matrix (CTM) - CTM controls the distribution of channel events. It provides Channel Interfaces for connection to either a CTI or CTM. This enables multiple ECTs to be connected to each other.

The figure below shows how CTIs and CTM are used in a generic setup.

Figure 34: CTIs and CTM in a Generic Setup



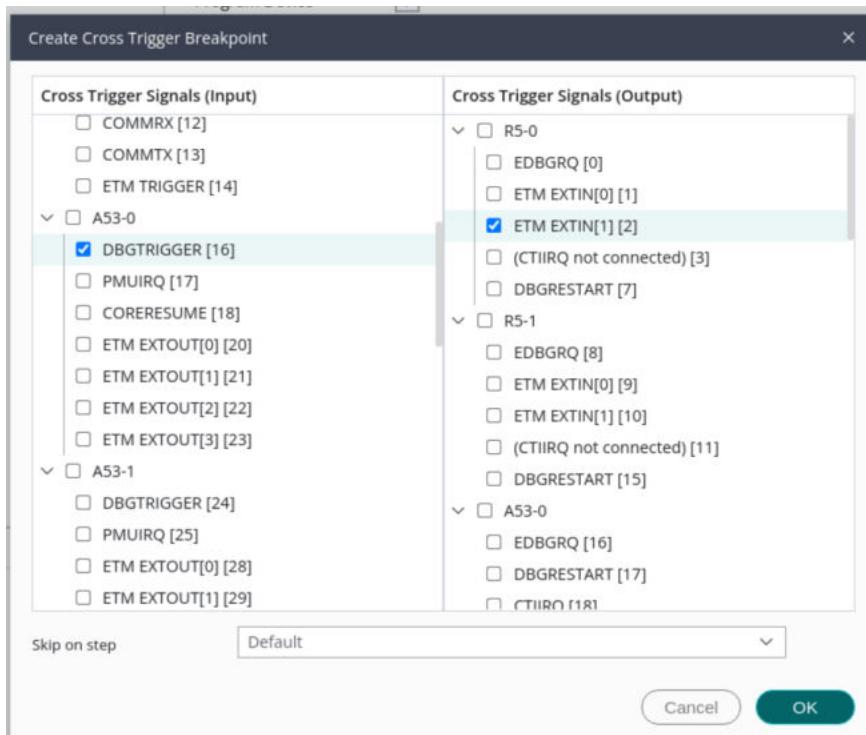
CTM forms an event broadcasting network with multiple channels. A CTI listens to one or more channels for an event, maps a received event into a trigger, and sends the trigger to one or more CoreSight components connected to the CTI. A CTI also combines and maps the triggers from the connected CoreSight components and broadcasts them as events on one or more channels. Through its register interface, each CTI can be configured to listen to specific channels for events or broadcast triggers as events to specific channels.

In the above example, there are four channels. The CTI at the top is configured to propagate the trigger event on Trigger Input 0 to Channel 0. Other CTIs can be configured to listen to this channel for events and broadcast the events through trigger outputs, to the debug components connected to these CTIs. CTIs also support channel gating such that selected channels can be turned off, without having to disable the channel to trigger I/O mapping.

Enable Cross-Triggering

You can create, edit, or remove cross-trigger breakpoints and apply the breakpoints on the target using the **Launch Configurations** page. To enable cross-triggering, do the following.

1. Launch the Vitis unified IDE.
2. Create a standalone application component and build it. Alternatively, you can select an existing application component.
3. Go to the **Flow Navigator** and select the application component.
4. See [Launch Configurations](#) and [Creating a New Target Connection](#) to launch the debug configuration and set the target connection.
5. Enable **Enable Cross Triggering** check-box in the launch configuration page. The **Cross Trigger Breakpoints** interface is displayed.
6. Click **Add Item** to create new breakpoints.



You can set the input and output signals or set the mode of **skip on step** on this screen.

Cross-Triggering in Zynq Devices

In Zynq devices, ECT is configured with four broadcast channels, four CTIs, and a CTM. One CTI is connected to ETB/TPIU, one to FTM and one to each Cortex-A9 core. The following table shows the trigger input and trigger output connections of each CTI.

Note: The connections specified in the table below are hard-wired connections.

Table 6: CTI Trigger Ports in Zynq Devices

CTI Trigger Port	Signal
CTI connected to ETB, TPIU	
Trigger Input 2	ETB full
Trigger Input 3	ETB acquisition complete
Trigger Input 4	ITM trigger
Trigger Output 0	ETB flush
Trigger Output 1	ETB trigger
Trigger Output 2	TPIU flush
Trigger Output 3	TPIU trigger
FTM CTI	
Trigger Input 0	FTM trigger
Trigger Input 1	FTM trigger
Trigger Input 2	FTM trigger
Trigger Input 3	FTM trigger
Trigger Output 0	FTM trigger
Trigger Output 1	FTM trigger
Trigger Output 2	FTM trigger
Trigger Output 3	FTM trigger
CPU0/1 CTIs	
Trigger Input 0	CPU DBGACK
Trigger Input 1	CPU PMU IRQ
Trigger Input 2	PTM EXT
Trigger Input 3	PTM EXT
Trigger Input 4	CPU COMMTX
Trigger Input 5	CPU COMMTX
Trigger Input 6	PTM TRIGGER
Trigger Output 0	CPU debug request
Trigger Output 1	PTM EXT
Trigger Output 2	PTM EXT
Trigger Output 3	PTM EXT
Trigger Output 4	PTM EXT
Trigger Output 7	CPU restart request

Cross-Triggering in Zynq UltraScale+ MPSoCs

In Zynq UltraScale+ MPSoCs, ECT is configured with four broadcast channels, nine CTIs, and a CTM. The table below shows the trigger input and trigger output connections of each CTI. These are hard-wired connections. For more details, refer to *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Table 7: CTI Trigger Ports in Zynq UltraScale+ MPSoCs

CTI Trigger Port	Signal
CTI 0 (soc_debug_fpd)	
IN 0	ETF 1 FULL
IN 1	ETF 1 ACQCOMP
IN 2	ETF 2 FULL
IN 3	ETF 2 ACQCOMP
IN 4	ETR FULL
IN 5	ETR ACQCOMP
IN 6	-
IN 7	-
OUT 0	ETF 1 FLUSHIN
OUT 1	ETF 1 TRIGIN
OUT 2	ETF 2 FLUSHIN
OUT 3	ETF 2 TRIGIN
OUT 4	ETR FLUSHIN
OUT 5	ETR TRIGIN
OUT 6	TPIU FLUSHIN
OUT 7	TPIU TRIGIN
CTI 1 (soc_debug_fpd)	
IN 0	FTM
IN 1	FTM
IN 2	FTM
IN 3	FTM
IN 4	STM TRIGOUTSPTE
IN 5	STM TRIGOUTSW
IN 6	STM TRIGOUTHETE
IN 7	STM ASYNCOUT
OUT 0	FTM
OUT 1	FTM
OUT 2	FTM
OUT 3	FTM
OUT 4	STM HWEVENTS
OUT 5	STM HWEVENTS
OUT 6	-
OUT 7	HALT SYSTEM TIMER
CTI 2 (soc_debug_fpd)	
IN 0	ATM 0
IN 1	ATM 1
IN 2	-
IN 3	-

Table 7: CTI Trigger Ports in Zynq UltraScale+ MPSoCs (cont'd)

CTI Trigger Port	Signal
IN 4	-
IN 5	-
IN 6	-
IN 7	-
OUT 0	ATM 0
OUT 1	ATM 1
OUT 2	-
OUT 3	-
OUT 4	-
OUT 5	-
OUT 6	-
OUT 7	picture debug start
CTI 0, 1 (RPU)	
IN 0	DBGTRIGGER
IN 1	PMUIRQ
IN 2	ETMEXTOUT[0]
IN 3	ETMEXTOUT[1]
IN 4	COMMRX
IN 5	COMMTX
IN 6	ETM TRIGGER
IN 7	-
OUT 0	EDBGRQ
OUT 1	ETMEXTIN[0]
OUT 2	ETMEXTIN[1]
OUT 3	-(CTIIRQ, not connected)
OUT 4	-
OUT 5	-
OUT 6	-
OUT 7	DBGRESTART
CTI 0, 1, 2, 3 (APU)	
IN 0	DBGTRIGGER
IN 1	PMUIRQ
IN 2	-
IN 3	-
IN 4	ETMEXTOUT[0]
IN 5	ETMEXTOUT[1]
IN 6	ETMEXTOUT[2]
IN 7	ETMEXTOUT[3]
OUT 0	EDBGRQ

Table 7: CTI Trigger Ports in Zynq UltraScale+ MPSoCs (cont'd)

CTI Trigger Port	Signal
OUT 1	DBGRESTART
OUT 2	CTIIRQ
OUT 3	-
OUT 4	ETMEXTIN[0]
OUT 5	ETMEXTIN[1]
OUT 6	ETMEXTIN[2]
OUT 7	ETMEXTIN[3]

Cross-Triggering in Versal Devices

In Versal devices, ECT is configured with four broadcast channels, 12 CTIs, and a CTM. The table below shows the trigger input and trigger output connections of each CTI. These are hard-wired connections. For more details, refer to the *Versal Adaptive SoC Technical Reference Manual* ([AM011](#)).

Table 8: CTI Trigger Ports in Versal Devices

CTI Trigger Port	Signal
R5 CTI 0,1 (RPU). XSDB IDs = 0-7 (R5 #0), 8-15 (R5 #1)	
IN 0	R5 DBGTRIGGER
IN 1	R5 PMUIRQ
IN 2	ETM EXTOUT[0]
IN 3	ETM EXTOUT[1]
IN 4	R5 COMMRX
IN 5	R5 COMMTX
IN 6	ETM TRIGGER
IN 7	-
OUT 0	R5 EDBGREQ
OUT 1	ETM EXTIN[0]
OUT 2	ETM EXTIN[1]
OUT 3	-
OUT 4	-
OUT 5	-
OUT 6	-
OUT 7	R5 DBGRESTART
CTI 0,1,2,3 (APU). XSDB IDs = 16-23 (A72 #0), 24-31 (A72 #1), 32-39 (A72 #2), 40-47 (A72 #3)	
IN 0	A72 DBGTRIGGER
IN 1	A72 PMUIRQ
IN 2	-

Table 8: CTI Trigger Ports in Versal Devices (cont'd)

CTI Trigger Port	Signal
IN 3	-
IN 4	ETM EXTOUT[0]
IN 5	ETM EXTOUT[1]
IN 6	ETM EXTOUT[2]
IN 7	ETM EXTOUT[3]
OUT 0	A72 EDBGREQ
OUT 1	A72 DBGRESTART
OUT 2	GIC PPI 24
OUT 3	-
OUT 4	ETM EXTIN[0]
OUT 5	ETM EXTIN[1]
OUT 6	ETM EXTIN[2]
OUT 7	ETM EXTIN[3]
CTI p (pmc_debug). XSDB IDs = 48-55	
IN 0	ATM TRIGOUT[0]
IN 1	-
IN 2	-
IN 3	-
IN 4	-
IN 5	-
IN 6	-
IN 7	-
OUT 0	ATM TRIGIN[0]
OUT 1	-
OUT 2	-
OUT 3	-
OUT 4	-
OUT 5	-
OUT 6	-
OUT 7	-
CTI 0d (soc_debug_ipd). XSDB IDs = 56-63	
IN 0	ATM0 TRIGOUT[0]
IN 1	ATM0 TRIGOUT[1]
IN 2	ATM0 TRIGOUT[2]
IN 3	ATM0 TRIGOUT[3]
IN 4	ATM0 TRIGOUT[4]
IN 5	-
IN 6	-
IN 7	-

Table 8: CTI Trigger Ports in Versal Devices (cont'd)

CTI Trigger Port	Signal
OUT 0	ATM0 TRIGIN[0]
OUT 1	ATM0 TRIGIN[1]
OUT 2	ATM0 TRIGIN[2]
OUT 3	ATM0 TRIGIN[3]
OUT 4	ATM0 TRIGIN[4]
OUT 5	7
OUT 6	-
OUT 7	-
CTI 1a (APU). XSDB IDs = 64-71	
IN 0	ELA 1a CTTRIGOUT[0]
IN 1	ELA 1a CTTRIGOUT[1]
IN 2	ETF 1a FULL
IN 3	ETF 1a ACQCOMP
IN 4	-
IN 5	-
IN 6	-
IN 7	-
OUT 0	ELA 1a CTTRIGIN[0]
OUT 1	ELA 1a CTTRIGIN[1]
OUT 2	ETF 1a FLUSHIN
OUT 3	ETF 1a TRIGIN
OUT 4	PMUSAPSHOT[0]
OUT 5	PMUSAPSHOT[1]
OUT 6	-
OUT 7	-
CTI 1b (soc_debug_fpd). XSDB IDs = 72-79	
IN 0	STM TRIGOUTSPTE
IN 1	STM TRIGOUTSW
IN 2	STM TRIGOUTHETE
IN 3	STM ASYNCOUT
IN 4	ETF 1 FULL
IN 5	ETF 1 ACQCOMP
IN 6	ETR FULL
IN 7	ETF ACQCOMP
OUT 0	STM HWEVENTS
OUT 1	STM HWEVENTS
OUT 2	TPIU FLUSHIN
OUT 3	TPIU TRIGIN
OUT 4	ETF 1 FLUSHIN

Table 8: CTI Trigger Ports in Versal Devices (cont'd)

CTI Trigger Port	Signal
OUT 5	ETF 1 TRIGIN
OUT 6	ETR FLUSHIN
OUT 7	ETR TRIGIN
CTI 1c (soc_debug_fpd). XSDB IDs = 80-87	
IN 0	pl_ps_trigger[0]
IN 1	pl_ps_trigger[1]
IN 2	pl_ps_trigger[2]
IN 3	pl_ps_trigger[3]
IN 4	-
IN 5	-
IN 6	-
IN 7	-
OUT 0	ps_pl_trigger[0]
OUT 1	ps_pl_trigger[1]
OUT 2	ps_pl_trigger[2]
OUT 3	ps_pl_trigger[3]
OUT 4	-
OUT 5	-
OUT 6	HALT System Timer
OUT 7	RESTART System Timer
CTI 1d (soc_debug_fpd). XSDB IDs = 88-95	
IN 0	ATM1 TRIGOUT[0]
IN 1	ATM1 TRIGOUT[1]
IN 2	ATM1 TRIGOUT[2]
IN 3	ATM1 TRIGOUT[3]
IN 4	ATM1 TRIGOUT[4]
IN 5	ATM1 TRIGOUT[5]
IN 6	ATM1 TRIGOUT[6]
IN 7	-
OUT 0	ATM1 TRIGIN[0]
OUT 1	ATM1 TRIGIN[1]
OUT 2	ATM1 TRIGIN[2]
OUT 3	ATM1 TRIGIN[3]
OUT 4	ATM1 TRIGIN[4]
OUT 5	ATM1 TRIGIN[5]
OUT 6	ATM1 TRIGIN[6]
OUT 7	-

Use Cases

FPGA to CPU Triggering

This is one of the most common use cases of cross-triggering in Zynq. There are four trigger inputs on FPGA CTI, which can be configured to halt (EDBGRQ) any of the two CPUs. Similarly, the four FPGA CTI trigger outputs can be triggered when a CPU is halted (DBGACK). The FPGA trigger inputs and outputs can be connected to ILA cores such that an ILA trigger can halt the CPU(s) and the ILA can be triggered to capture the signals its monitoring, when any of the two CPUs is halted. For more details about setting up cross-triggering to the FTM in Vivado Design Suite, refer to the Cross Trigger Design section in *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* ([UG940](#)).

PTM to CPU Triggering

Synchronize trace capture with the processor state. For example, an ETB full event can be used as a trigger to halt the CPU(s).

CPU to CPU Triggering

Cross-triggering can be used to synchronize the entry and exit from debug state between the CPUs. For example, when CPU0 is halted, the event can be used to trigger a CPU1 debug request, which can halt CPU1.

XSCT Cross-Triggering Commands

The XSCT breakpoint add command (bpadd) has been enhanced to enable cross triggering between different components.

For example, use the following command to set a cross trigger to stop Zynq core 1 when core 0 stops.

```
bpadd -ct-input 0 -ct-output 8
```

For Zynq, -ct-input 0 refers to CTI CPU0 TrigIn0 (trigger input 0 of the CTI connected to CPU0), which is connected to DBGACK (asserted when the core is halted). -ct-output 8 refers to CTI CPU1 TrigOut0, which is connected to CPU debug request (asserting this pin halts the core). hw_server uses an available channel to set up a cross trigger path between these pins. When core 0 is halted, the event is broadcast to core 1 over the selected channel, causing core 1 to halt.

Use the following command for the Zynq UltraScale+ MPSoC to halt the A53 core 1 when A53 core 0 stops.

```
bpadd -ct-input 16 -ct-output 24
```

Profile/Analyze

TCF Profiling

The TCF profiler supports profiling of both standalone and Linux applications. TCF profiling does not require any additional compiler flags to be set while building the application. Profiling standalone applications over JTAG is based on sampling the Program Counter through debug interface. It does not alter the program execution flow and is non-intrusive when stack trace is not enabled. When stack trace is enabled, program execution speed decreases as the debugger has to collect stack trace information.

1. Start the Vitis Unified IDE and select the application component you wish to profile.
2. Go to the Flow Navigator and create a debug launch configuration.

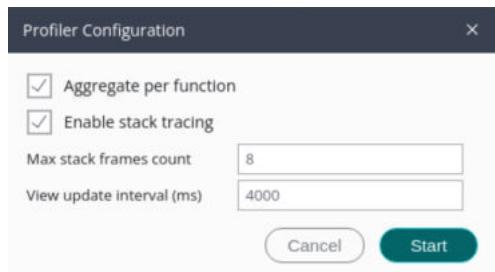
Note: Refer to [Launch Configurations](#) and [Creating a New Target Connection](#) to create a debug launch configuration.

Note: Regarding the target connection, if the application component is a Linux application, select the TCF agent. If it a standalone application, select HW server.

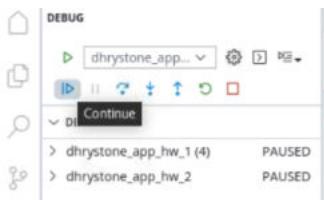
3. After configuration is finished, click **Debug** to start the debug session.
Note: Refer to [Starting Debug](#) to start the debug session.
4. Click the **TCF Profiler** button to start profiling view.



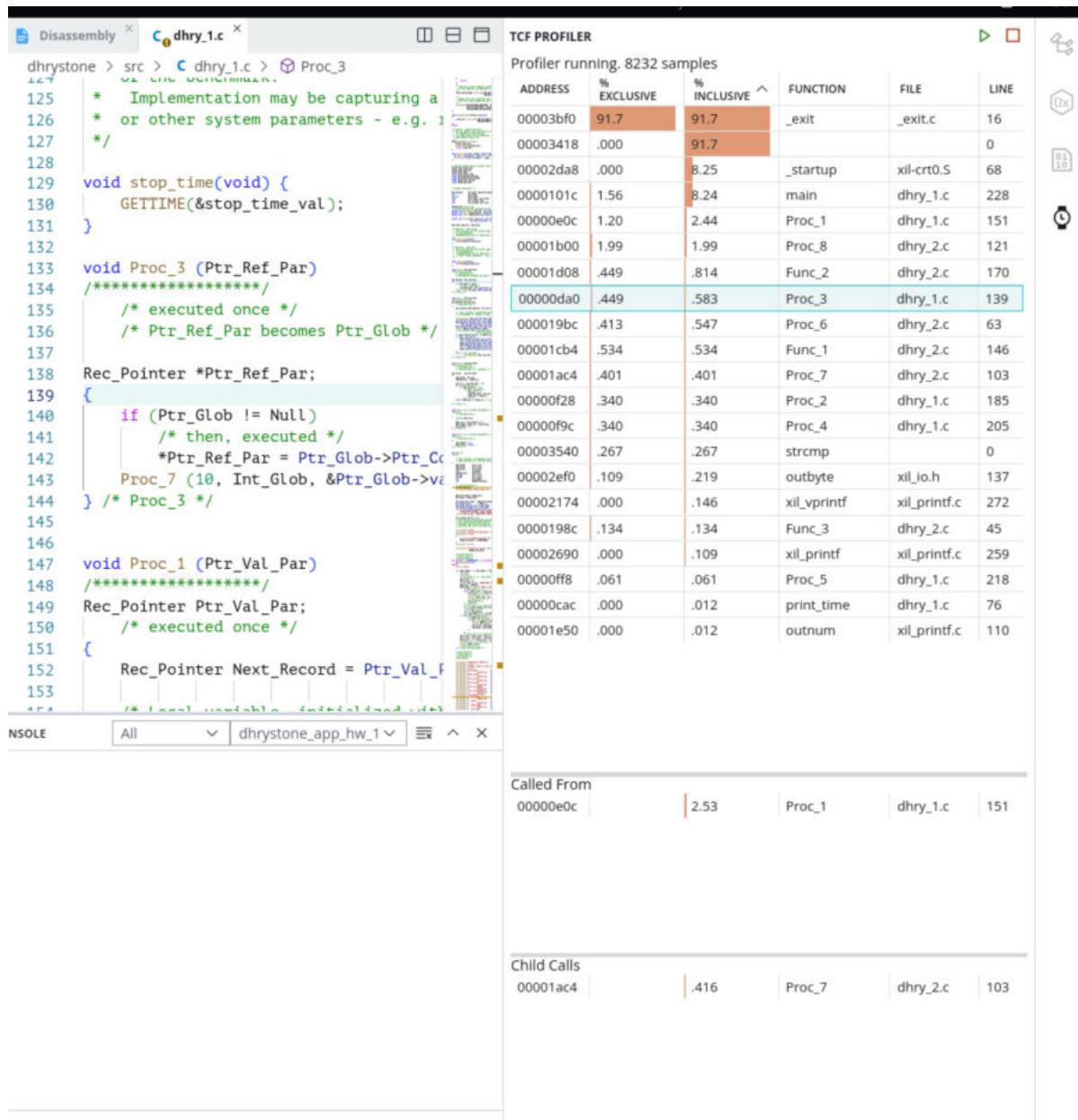
5. Click the **Start** button to begin the profiling. Select the **Enable stack tracing** option to show the stack trace for each address in the sample data. You can also set the frame count and update the interval according to your requirements.
 - Specify the **Max stack frames count** for the maximum number of frames that are shown in the stack trace view.
 - Specify the **View update interval** for the time interval (in milliseconds) the TCF profiler view is updated with the new results. Ensure that this is different from the interval at which the profile samples are collected.



6. Click **Start** to profiling.
7. Click **Continue** to free run the application.



8. You can now retrieve the profiling information of your application component. The profiling data is displayed:



- Clicking the function in the profiler view displays the Called From and Child Calls in the bottom right corner of the Profiler view.
- The view supports cross-probe between the function name and source code. Clicking the function jumps to the source code in the Source Code view.

gprof Profiling

Note: This function is not supported in the Vitis Unified IDE. Switch to classic Vitis IDE if you wish to access this option.

Non-Intrusive Profiling for MicroBlaze Processors

This function is not supported in the Vitis Unified IDE. Switch to classic Vitis IDE and refer to 2023.1 documentation, if you wish to access this option.

FreeRTOS Analysis using STM

Note: This function is not supported in the Vitis Unified IDE. Change to the classic IDE if you want to use this function.

Optimize: Performance Analysis

Note: This function is not supported in the Vitis Unified IDE in this version. Change to the classic IDE if you want to use this function. Refer to 2023.1 documentation for details about how to use this feature in Classic Vitis IDE.

Creating a Boot Image

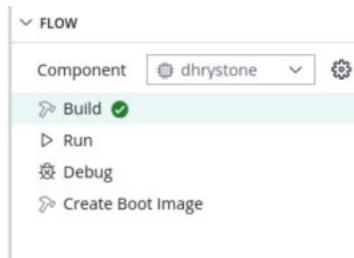
AMD FPGAs and system-on-chip (SoC) devices typically have multiple hardware and software binaries used to boot them to the function they were designed for. These binaries can include FPGA bitstreams, firmware images, bootloaders, operating systems, and user-chosen applications that can be loaded in both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

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

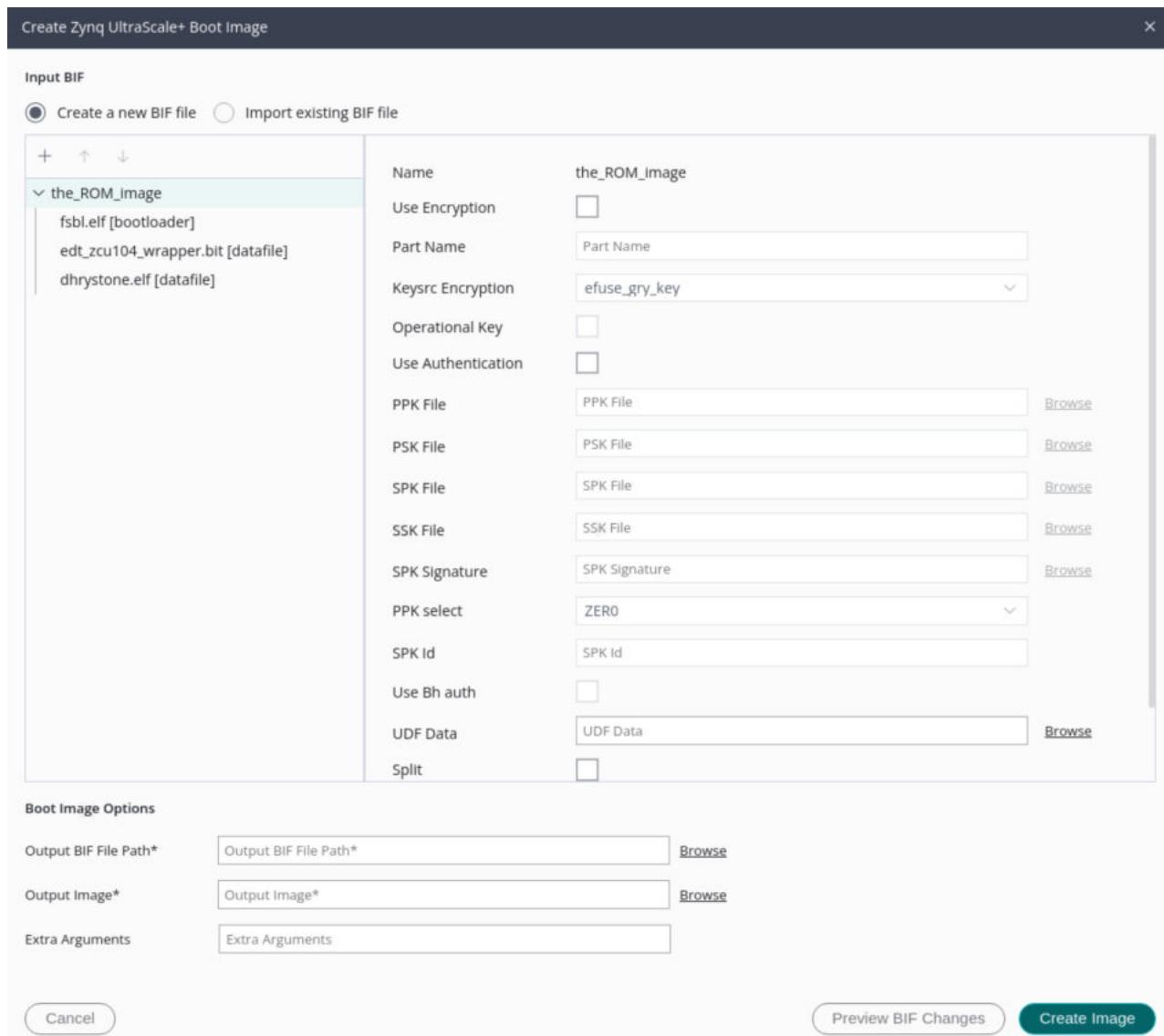
There are two ways to create a boot image: from the Flow Navigator or from scratch.

Create boot image from Flow Navigator (faster)

1. Select the application component which has already been built in the Component view.
2. Go to the Flow Navigator and click **Create Boot Image**.



3. The create boot image setup dialog is displayed. It is populated with the required images from your component. Specify the **Output Bif File Path** and **Output Image path**.



4. Click **Create Image**. After a few seconds, the log should indicate that the created image is ready.

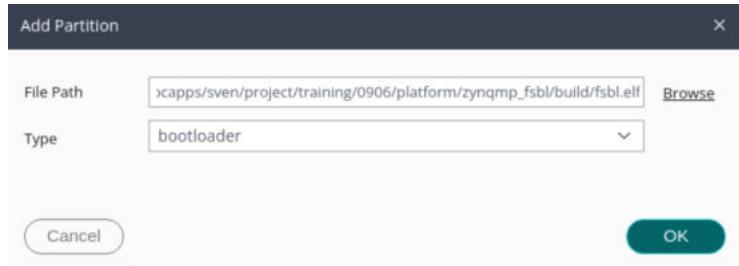
Create a boot image from scratch

After your application component compilation is finished, follow the steps below to create the boot image.

From **Vitis** → **Create Boot Image** and select the device according to your design. The Create Image wizard is displayed.

ZYNQ/ZYNQMP

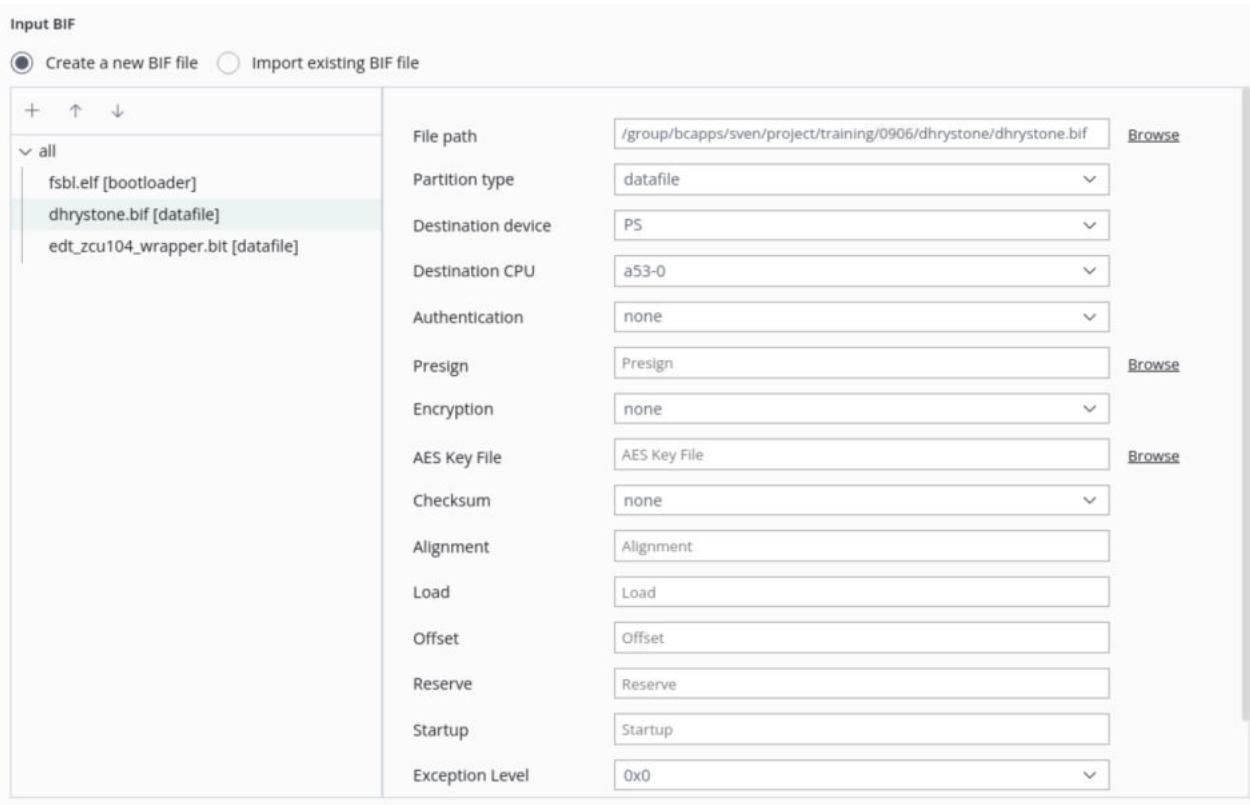
1. If you do not have an existing bif file, select Create a new BIF file.
2. Click + to add the image partition.



3. Specify the image, set the Type and click **OK**.

Note: You can add the boot loader, FPGA bit file, bl31 and, application ELF file according to your requirements.

Note: You can modify the image attribute by selecting the image (see following example) and change the attributes according to your requirements.



4. Specify the Output Bif File Path and Output Image path.
5. Click **Create Image**. After a few seconds, the log should indicate the created image is ready.

For more information about the Bootgen utility, refer to the *Bootgen User Guide* ([UG1283](#)).

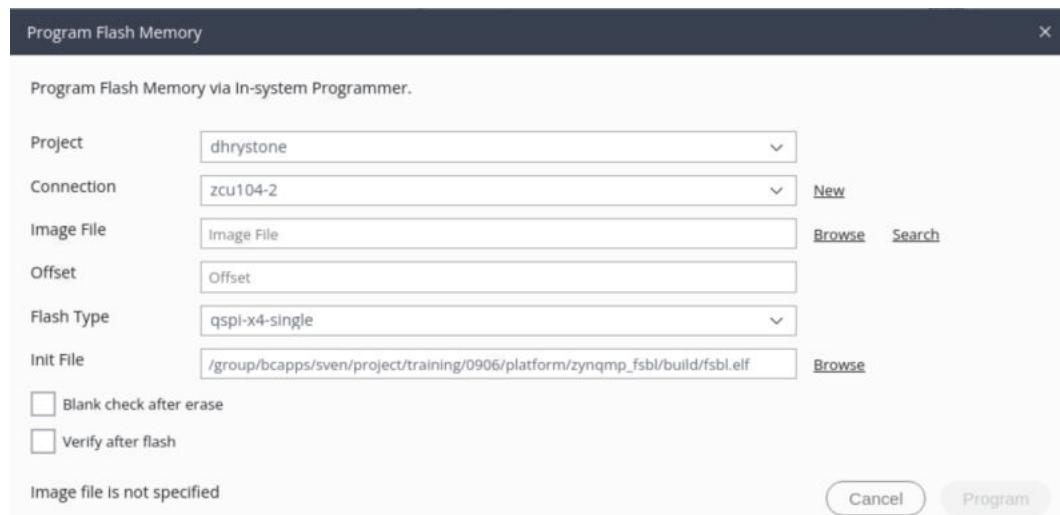
Programming Flash

Program Flash is a tool used to program flash memories in the design. Various types of flash are supported for programming.

- For non-Zynq devices – Parallel Flash (BPI) and Serial Flash (SPI) from various makes such as Micron and Spansion.
- For Zynq devices – QSPI, NAND, and NOR. QSPI can be used in different configurations such as QSPI SINGLE, QSPI DUAL PARALLEL, and QSPI DUAL STACKED.
- For Versal devices – QSPI, emmc, and OSPI. QSPI can be used in different configurations such as QSPI SINGLE, QSPI DUAL PARALLEL, and QSPI DUAL STACKED.

Go to **Vitis** → **Program Flash** in the menu and open the program flash wizard.

Figure 35: Program Flash Window



The options available on the **Program Flash Memory** page are as follows:

- **Project:** Select the system project you plan to use. The application component is be automatically selected in the Component View.
- **Connection:** Select the connection to the hardware server.
- **Image File:** Select the file to write to the flash memory.
 - Zynq devices:
 - Supported file formats for qspi flash types are BIN or MCS formats.
 - Supported file formats for nor and nand types is BIN format.
 - Non-Zynq devices:

- Supported types for flash parts in non Zynq devices are BIT, ELF, SREC, MCS, BIN.
- **Offset:** Specify the offset relative to the Flash Base Address, where the file should be programmed.
Note: Offset is not required for MCS files.
- **Flash Type:** Select a flash type.
 - Zynq devices:
 - qspi_single
 - qspi_dual_parallel
 - qspi_dual_stacked
 - nand_8
 - nand_16
 - nor
 - emmc
- **Non-Zynq devices:**
 - The flash type drop-down list is populated based on the FPGA detected in the connection. If the connection to the hardware server does not exist, an error message stating "Could not retrieve Flash Part information. Please check hardware server connection" is displayed on the page. Based on the device detected, the dialog populates all the flash parts supported for the device.

Note: The appropriate part can be selected based on the design. For AMD boards, the part name can be found from the respective board's user guide.

- **Init File:** Provide the initialization file path.
- **Blank check after erase:** The blank check is performed to verify if the erase operation was properly done. The contents are read back and checked if the region erased is blank.
- **Verify after Flash:** The verify operation is cross-checked with the flash programming operation. The flash contents are read back and cross-checked against the programmed data.

Multi-Cable and Multi-Device Support

This feature of the Vitis environment allows you to run and debug application projects, program the bitstream/PDI, and program the flash using multiple JTAG cables or multiple devices on a single JTAG chain. The main use cases are as follows:

- **Multi-cable:** You have more than one board connected to the system and you want to work on all of the boards.
- **Multi-device:** You have multiple devices in a JTAG chain and you want to work all of the devices.

In the Vitis tool, the following operations are relevant to these use cases:

- Target connection (view only)
- Debugging applications using System Debugger (you can also program the FPGA during debug)
- Programming the FPGA
- Programming the flash

Viewing Target Connections

The following snapshots have been achieved with the help of Vitis Unified IDE. This view displays the device information associated with the tool.

Figure 36: Target Connections Details Window

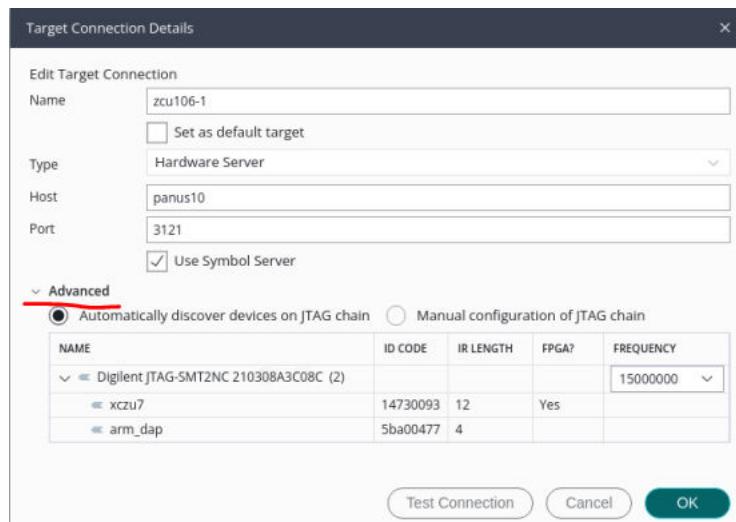


Figure 37: Figure: Multi-Cable Target Connections

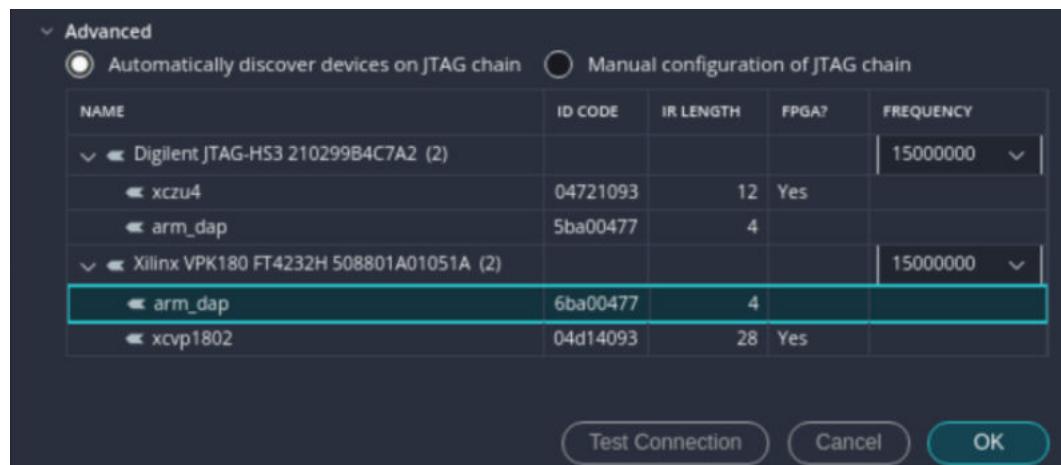
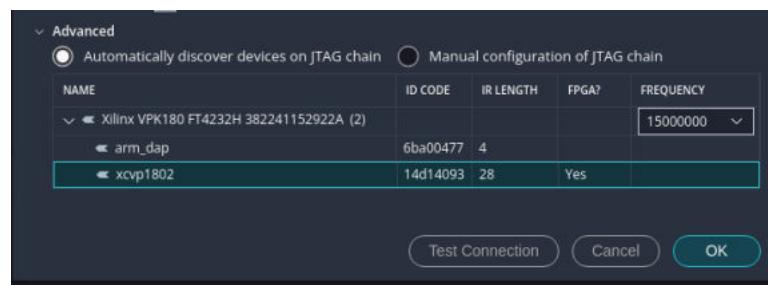


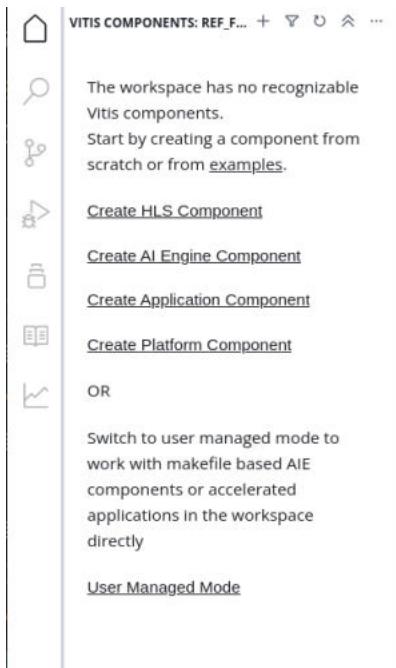
Figure 38: Figure: Multi-Device Target Connections



User Managed Flow

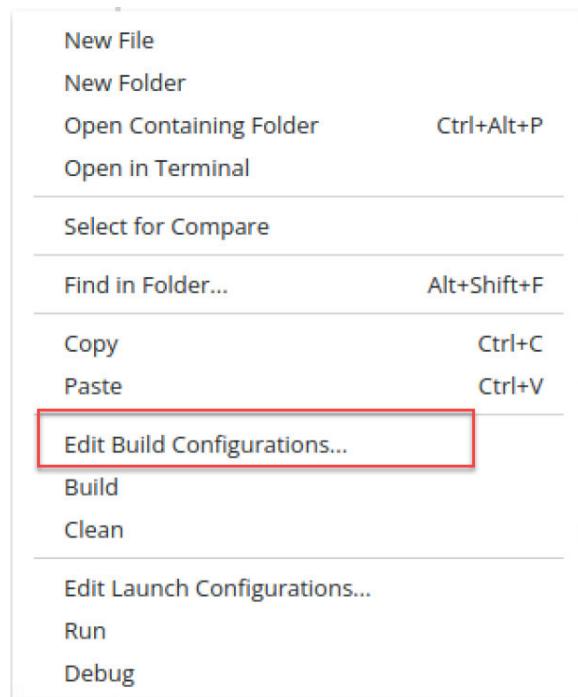
If you prefer to manage your source code hierarchy and build process in a unique way, for instance, by crafting your own Makefile or CMake files, you can use User Managed Mode. This mode empowers you by offering enhanced support and debugging capabilities, especially when you oversee the build flow. Additionally, the User Managed Flow simplifies the process, making the build launch more seamless and user-friendly. If you prefer command line flow to develop your application, User managed flow is an ideal choice. This flow supports Makefile based projects. Execute the following steps to use the flow.

1. Start Vitis Unified IDE.
2. Open a workspace where Makefile based project exists.
3. Click **User Managed Flow** to switch to user managed flow in the Component view. The Explorer view must contains all the files in the directory.

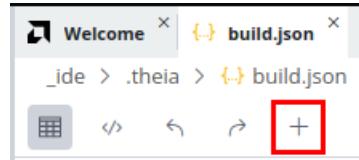


Note: The explorer view can also be reached from **Menu → View → Explorer**.

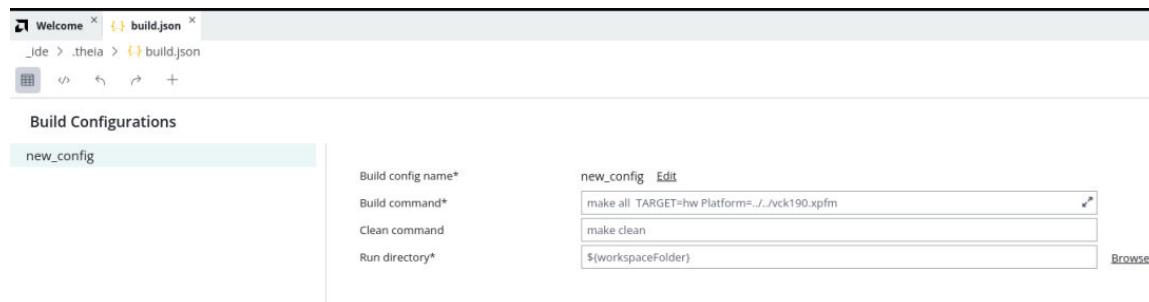
4. Create build configuration.
 - a. Right click anywhere on **File Explorer** → **Edit Build Configuration**.



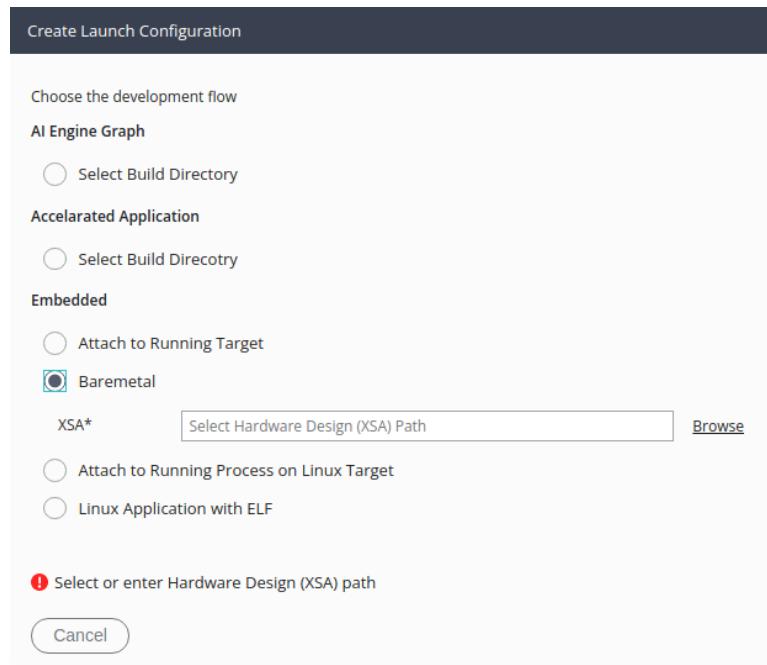
- b. Create a new build configuration by clicking + button,



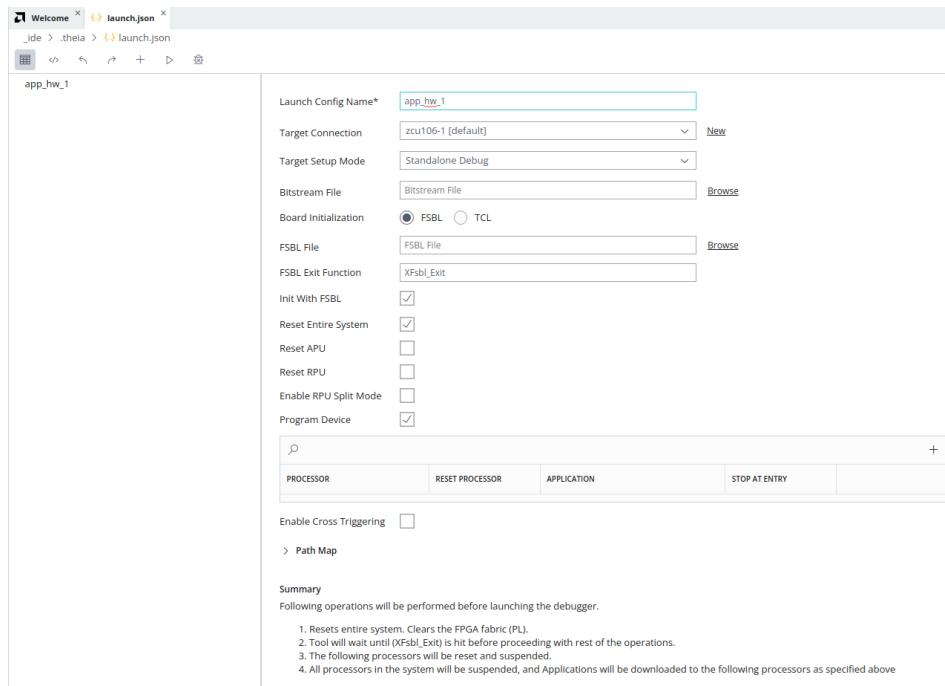
- c. A new configuration is created. Input your build command and clean command. You can also specify the build directory, if needed. See the following figure for an example.



- d. Right click anywhere in the file explorer and select **Build** to build your project.
5. Create Run/Debug launch configuration after build is finished.
- Right click anywhere in the file explorer and select **Edit Launch Configuration**.
 - Create a launch configuration by clicking + button.
 - It brings up Create Launch Configuration wizard. You have four options for embedded application component.



- Attach to Running Target: If you desire to debug on the already running target, select this option. You need to provide the target connection and begin to debug.



Note: Refer to [Target Connections](#) for creating the target connections.

- Baremetal: This is used to debug or run the bare-metal application component. Specify the XSA file and click **Submit** to create the configuration file. Wait until the Launch Configuration file pops up. Refer to [Launch Configurations](#) for setting references.

- iii. Attach to Running Process on Linux Target: To debug a Linux application on an already running Linux target, select this option and click **Submit**. Launch configuration window appears. Provide the target connection and executable file, post which you can run/debug it.
 - iv. Linux Application with ELF: If a Linux application has to be run/debugged, select this option and provide the elf. Launch configuration with the few sections is created for the app. Provide the necessary information post which you can run/debug it.
-

Setting User Specified Tool Chain

Tool supports multiple versions of toolchain in user managed mode. To do this, execute the following steps.

1. Open terminal in Linux system or open windows command prompt.
2. In the command line, input the following commands.

```
setenv VITIS_IDE_USER_TOOLCHAIN <custom_tool_chain_bin_directory_path>
```

You can start Vitis from the terminal and change to user managed mode.

Vitis Utilities

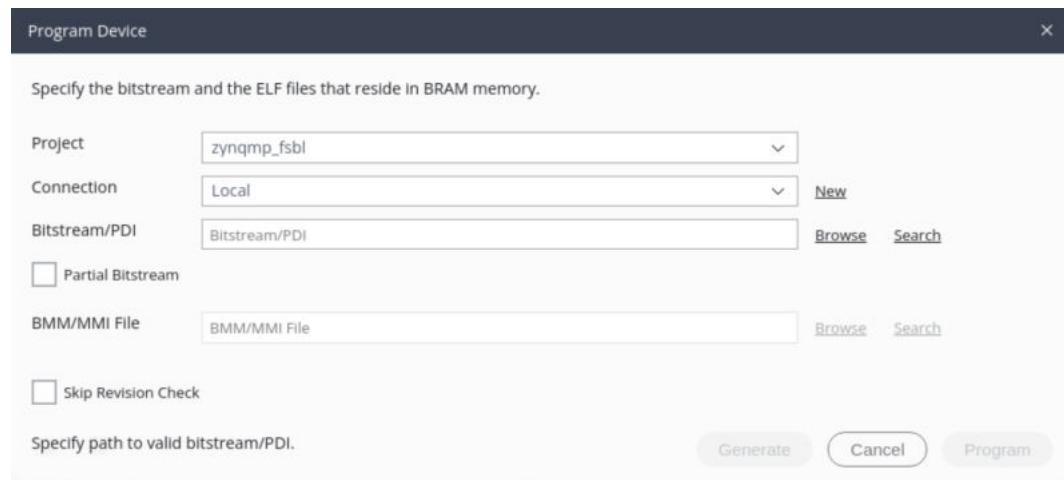
Software Command-Line Tool

Launch a console window to interact with XSCT. For more information on XSCT, see the [Software Command-Line Tool](#) section and the [XSCT Commands](#) section.

Program Device

Program the FPGA with the bitstream. From Vitis **Program Device**, you can get the following setup dialog of programming device.

Figure 39: Program Device



The following table lists the options available on the Program Device page:

- **Project:** Select the system project you want to use.
- **Connection:** Select the hardware or hardware server if the board is on remote side.
- **Bitstream/PDI File:** Specify the bitstream/PDI file

- **Partial Bitstream:** Indicate this is a partial bit stream file.
 - **BMM/MMI file:** Only for MicroBlaze design. These files store the BRAM name and location information of MicroBlaze. They are generated by Vivado.
 - **Skip Revision Check:** Skip version check
 - **Generate:** Stitch the elf with the bit stream and generate the final bit stream file
 - **Program:** Click this button to program the FPGA.
-

Vitis Terminal

From **Terminal** → **New Terminal** you can create a new terminal. This terminal is forked from current working environment. This terminal can be used for running any AMD standalone utility (Bootgen, Program Flash, Program device, and so on).

Project Export and Import

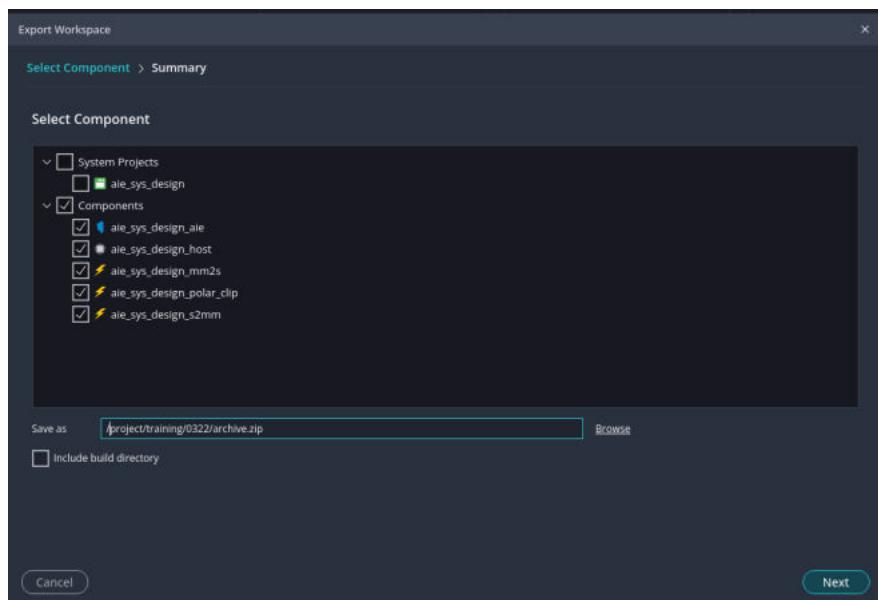
The AMD Vitis™ Unified IDE provides a simplified method for exporting or importing one or more AMD Vitis™ Unified IDE projects or components within your workspace.

Export Vitis Component

When exporting a component, the component is archived in a zip file with all the relevant files needed to import to another workspace.

1. To export a component, select **File** → **Export** from the main menu.

The Vitis Export Workspace wizard opens, where you can select the component or components in the current workspace to export as shown in the following figure.



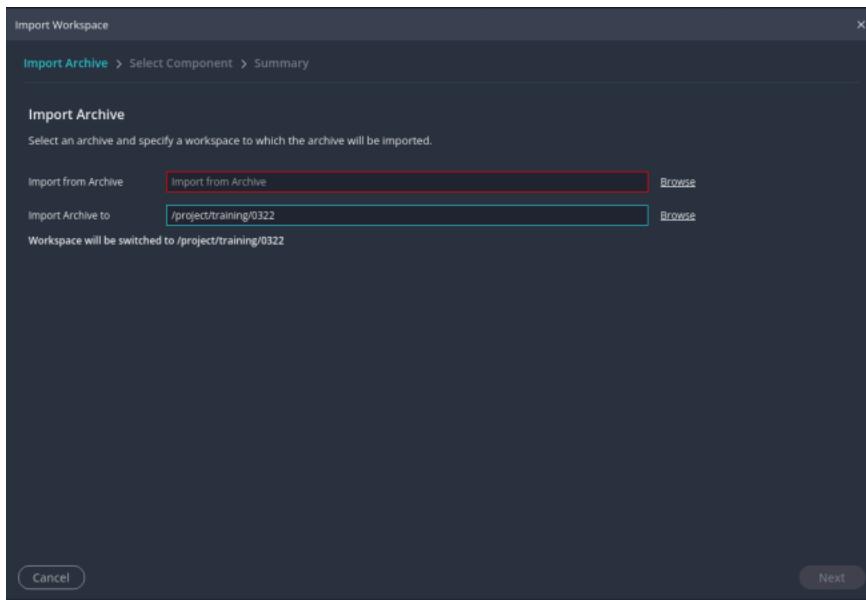
2. Select the component you want to export
3. Click **Browse** to select a location to save the Archive file. Then, click **Next**.
4. Review the summary of the exported components and click **Finish**.

Note: The selected Vitis Unified IDE components are archived in the specified file and location, and can be imported into the Vitis Unified IDE under a different workspace, on a different computer, by a different user.

Import Vitis Component

1. To import a project, select **File → Import** from the top menu.

The Vitis Import Workspace wizard opens, where you can select the archive file and specify the new workspace as shown in the following figure.



2. Select the archive file to be imported and specify the new workspace and click **Next**.
3. Select the component you want to import to your workspace in this page and click **Next**.
4. Review the summary of the imported components and click **Finish**.

Generating Device Tree

Note: This function is not supported in the Vitis Unified IDE in this version. Change to the classic IDE if you want to use this function. Refer to 2023.1 documentation for details about how to use this feature in Classic Vitis IDE.

Bootgen Tool

This section contains the following chapters:

- [Introduction](#)
- [Boot Image Layout](#)
- [Creating Boot Images](#)
- [Using Bootgen GUI](#)
- [Boot Time Security](#)
- [FPGA Support](#)
- [Use Cases and Examples](#)
- [BIF Attribute Reference](#)
- [Command Reference](#)
- [CDO Utility](#)

Introduction

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

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

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

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

Installing Bootgen

You can use Bootgen in GUI mode in the Vitis IDE for simple boot image creation, or in command-line mode for more complex boot images. You can install Bootgen from the AMD Unified Installer for FPGAs and Adaptive SoCs.

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

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

Boot Time Security

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

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

- [Zynq 7000 SoC Technical Reference Manual \(UG585\)](#)
- [Zynq UltraScale+ Device Technical Reference Manual \(UG1085\)](#)
- [Versal Adaptive SoC Technical Reference Manual \(AM011\)](#)

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

See [Using Encryption](#) and [Using Authentication](#) for more information about encrypting and authenticating content when using Bootgen.

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

Boot Image Layout

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

- For information about using Bootgen for Zynq 7000 devices, see [Zynq 7000 SoC Boot and Configuration](#).
- For information about using Bootgen for AMD Zynq™ UltraScale+™ MPSoC devices, see [Zynq UltraScale+ MPSoC Boot and Configuration](#).
- For information on how to use Bootgen for AMD FPGAs, see [FPGA Support](#).
- For information on AMD Versal™ adaptive SoC, see [Versal Adaptive SoC Boot Image Format](#).

Building a boot image involves the following steps:

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

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

The input files are not necessarily different for each device (for example, for every device, ELF files can be input files that can be part of the boot image), but the format of the boot image is different. The following sections describe the required format of the boot header, image header, partition header, initialization, and authentication certificate header for each device.

Zynq 7000 SoC Boot and Configuration

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

BootROM on Zynq 7000 SoC

The BootROM is the first software to run in the application processing unit (APU). BootROM executes on the first Cortex® processor, A9-0, while the second processor, Cortex, A9-1, executes the wait for event (WFE) instruction. The main tasks of the BootROM are to configure the system, copy the FSBL from the boot device to the on-chip memory (OCM), and then branch the code execution to the OCM.

Optionally, you can execute the FSBL directly from a Quad-SPI or NOR device in a non-secure environment. The master boot device holds one or more boot images. A boot image is made up of the boot header and the first stage boot loader (FSBL). Additionally, a boot image can have programmable logic (PL), a second stage boot loader (SSBL), and an embedded operating system and applications; however, these are not accessed by the BootROM. The BootROM execution flow is affected by the boot mode pin strap settings, the boot header, and what it discovers about the system. The BootROM can execute in a secure environment with encrypted FSBL, or a non-secure environment. The supported boot modes are:

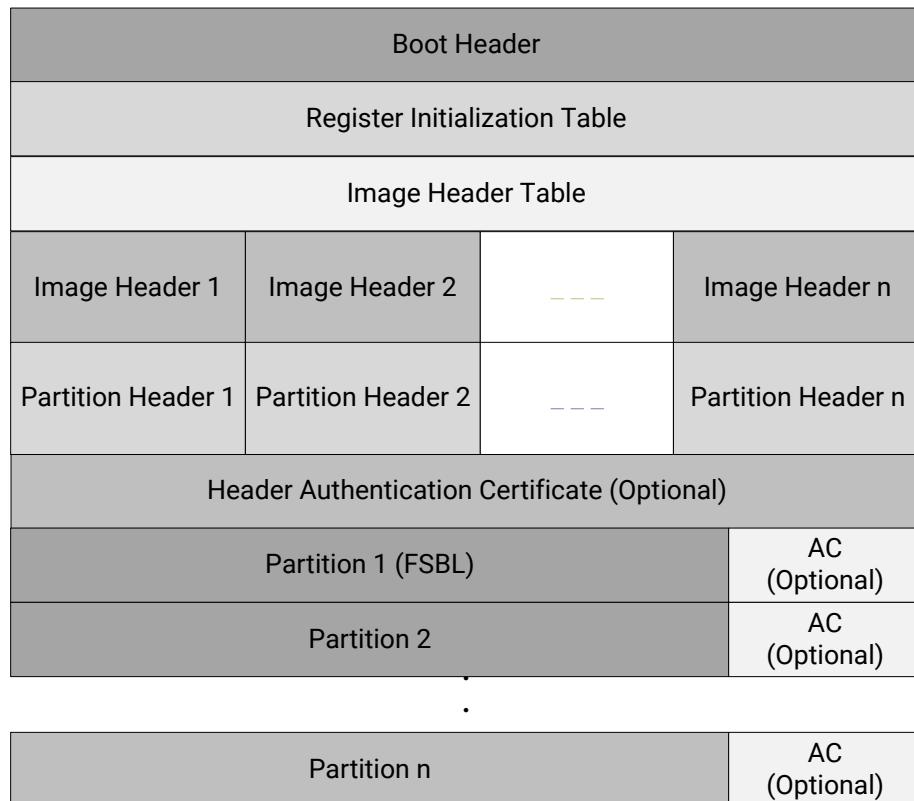
- JTAG mode is primarily used for development and debug.
- NAND, parallel NOR, Serial NOR (Quad-SPI), and secure digital (SD) flash memories are used for booting the device.

The *Zynq 7000 SoC Technical Reference Manual* ([UG585](#)) provides the details of these boot modes. See [Answer Record 52538](#) for answers to common boot and configuration questions.

Zynq 7000 SoC Boot Image Layout

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

Figure 40: Boot Header



X25912-102521

Zynq 7000 SoC Boot Header

Additionally, the Boot Header contains a [Zynq 7000 SoC Register Initialization Table](#). BootROM uses the boot header to find the location and length of FSBL and other details to initialize the system before handing off the control to FSBL.

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

The additional boot image components are:

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

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

Table 9: Zynq 7000 SoC Boot Header

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

Zynq 7000 SoC Register Initialization Table

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

Table 10: Zynq 7000 SoC Register Initialization Table

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

Zynq 7000 SoC Image Header Table

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

Table 11: Zynq 7000 SoC Image Header Table

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

Zynq 7000 SoC Image Header

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

Table 12: Zynq 7000 SoC Image Header

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

Zynq 7000 SoC Partition Header

The Partition Header is an array of structures containing information related to each partition. Each partition header table is parsed by the Boot Loader. The information such as the partition size, address in flash, load address in RAM, encrypted/signed, and so forth, are part of this table. There is one such structure for each partition including FSBL. The last structure in the table is marked by all `NULL` values (except the checksum). The following table shows the offsets, names, and notes regarding the AMD Zynq™ 7000 SoC Partition Header.

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

Table 13: Zynq 7000 SoC Partition Header

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

Table 13: Zynq 7000 SoC Partition Header (cont'd)

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

Zynq 7000 SoC Partition Attribute Bits

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

Table 14: Zynq 7000 SoC Partition Attribute Bits

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

Zynq 7000 SoC Authentication Certificate

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

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

The format of the Authentication Certificate in an AMD Zynq™ 7000 SoC is as shown in the following table.

Table 15: Zynq 7000 SoC Authentication Certificate

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

Zynq 7000 SoC Authentication Certificate Header

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

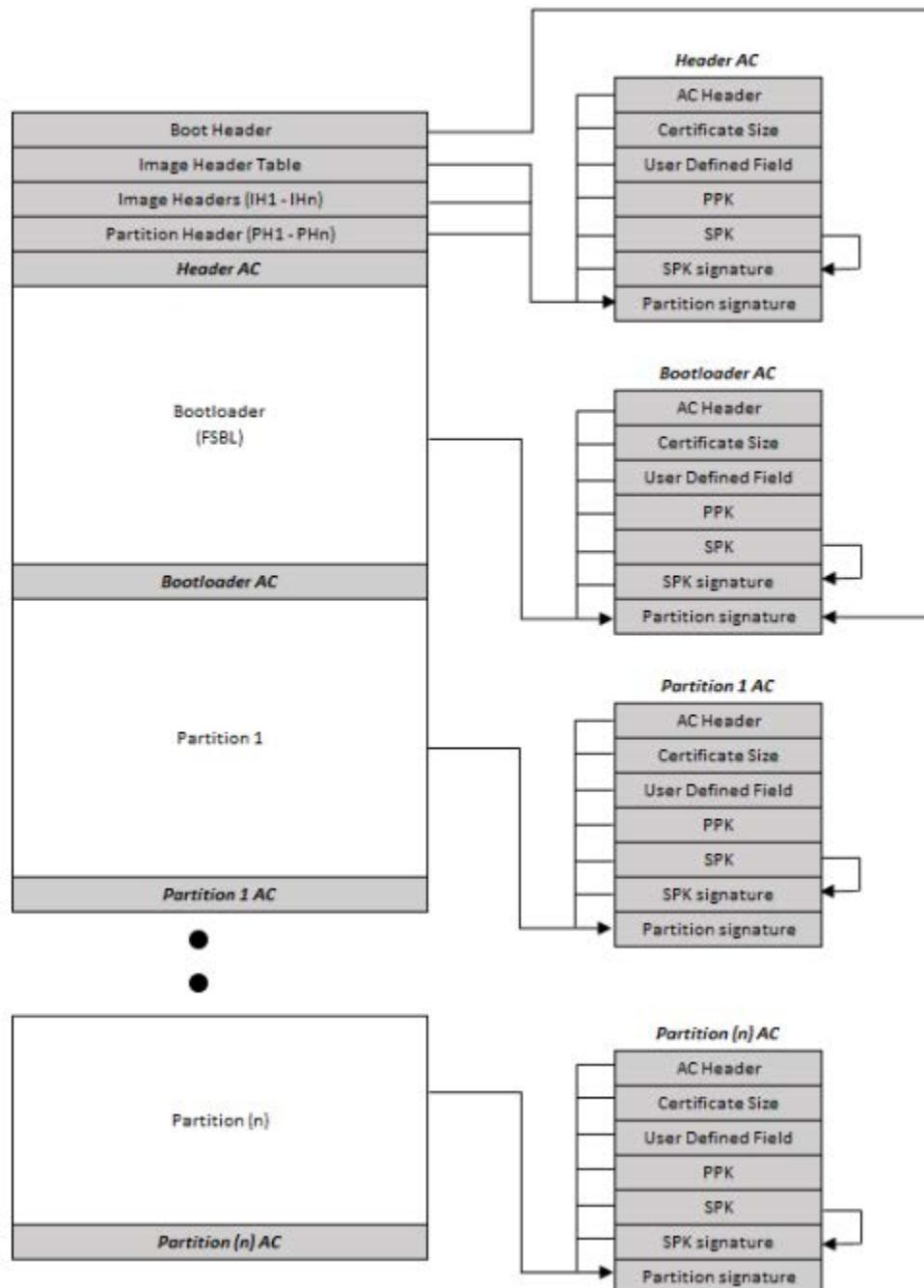
Table 16: Zynq 7000 SoC Authentication Certificate Header

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

Zynq 7000 SoC Boot Image Block Diagram

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

Figure 41: Zynq 7000 SoC Boot Image Block Diagram



Zynq UltraScale+ MPSoC Boot and Configuration

Introduction

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

During initial boot, the following steps occur:

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

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

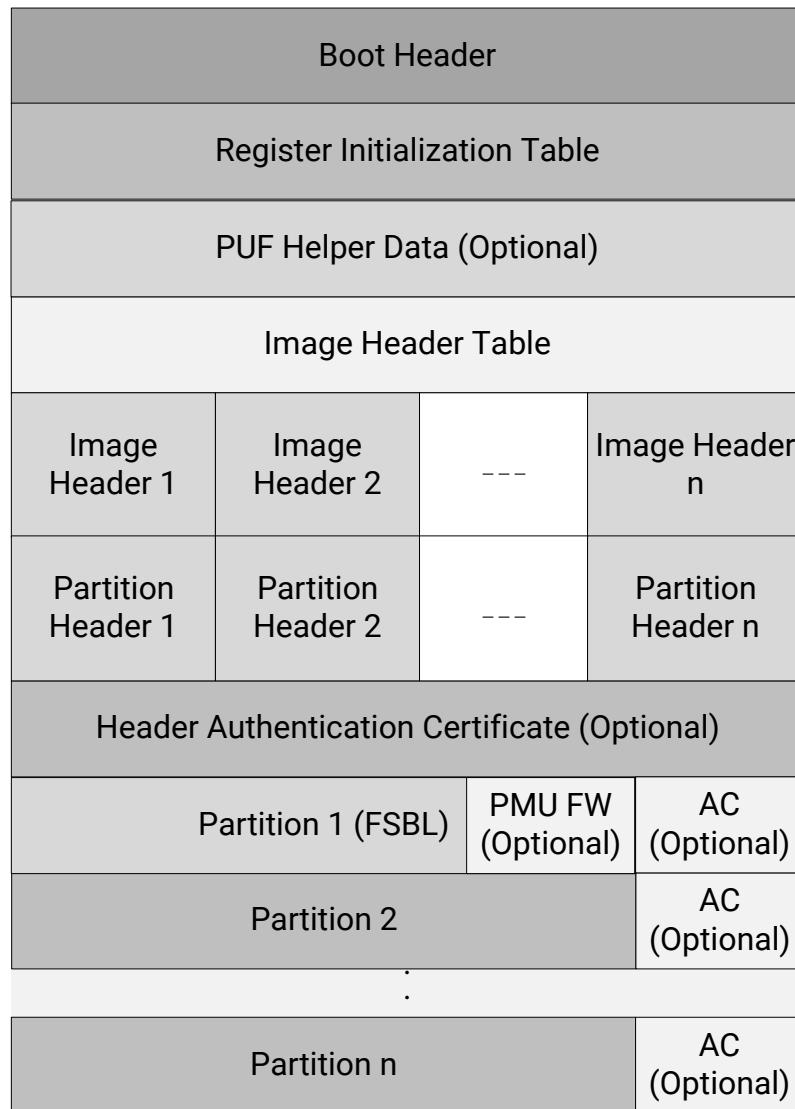
- Checks to determine if authentication is required by the FSBL or the user application.
- Performs an authentication check and proceeds only if the authentication check passes. Then checks the image for any encrypted partitions.
- If the CSU detects partitions that are encrypted, the CSU performs decryption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

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

Zynq UltraScale+ MPSoC Boot Image

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

Figure 42: Zynq UltraScale+ MPSoC Boot Image



X23449-102919

Zynq UltraScale+ MPSoC Boot Header

About the Boot Header

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

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

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

Table 17: Zynq UltraScale+ MPSoC Device Boot Header

Address Offset	Parameter	Description
0x00-0x1F	Arm® vector table	XIP ELF vector table: 0xEFFFFFFE: for Cortex®-R5F and Cortex A53 (32-bit) 0x14000000: for Cortex A53 (64-bit)
0x20	Width Detection Word	This field is used for QSPI width detection. 0xAA995566 in little endian format.
0x24	Header Signature	Contains 4 bytes 'X', 'N', 'L', 'X' in byte order, which is 0x584c4e58 in little endian format.
0x28	Key Source	0x00000000 (Un-Encrypted) 0xA5C3C5A5 (Black key stored in eFUSE) 0xA5C3C5A7 (Obfuscated key stored in eFUSE) 0x3A5C3C5A (Red key stored in BBRAM) 0xA5C3C5A3 (eFUSE RED key stored in eFUSE) 0xA35C7CA5 (Obfuscated key stored in Boot Header) 0xA3A5C3C5 (USER key stored in Boot Header) 0xA35C7C53 (Black key stored in Boot Header)
0x2C	FSBL Execution address (RAM)	FSBL execution address in OCM or XIP base address.
0x30	Source Offset	If no PMUFW, then it is the start offset of FSBL. If PMUFW, then start of PMUFW.
0x34	PMU Image Length	PMU firmware original image length in bytes. (0-128KB). If size > 0, PMUFW is prefixed to FSBL. If size = 0, no PMUFW image.
0x38	Total PMU FW Length	Total PMUFW image length in bytes.(PMUFW length + encryption overhead)

Table 17: Zynq UltraScale+ MPSoC Device Boot Header (cont'd)

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

Zynq UltraScale+ MPSoC Boot Header Attribute Bits

Table 18: Zynq UltraScale+ MPSoC Boot Header Attribute Bits

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

Table 18: Zynq UltraScale+ MPSoC Boot Header Attribute Bits (cont'd)

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

Zynq UltraScale+ MPSoC Register Initialization Table

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

Table 19: Zynq UltraScale+ MPSoC Register Initialization Table

Address Offset	Parameter	Description
0xB8 to 0x8B4	Register Initialization Pairs: <address>:<value>: (2048 bytes)	If the Address is set to 0xFFFFFFFF, that register is skipped and the value is ignored. All unused register fields must be set to Address=0xFFFFFFFF and value =0x0.

Zynq UltraScale+ MPSoC PUF Helper Data

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

Table 20: Zynq UltraScale+ MPSoC PUF Helper Data

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

Zynq UltraScale+ MPSoC Image Header Table

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

Table 21: Zynq UltraScale+ MPSoC Device Image Header Table

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

Zynq UltraScale+ MPSoC Image Header

About Image Headers

The Image Header is an array of structures containing information related to each image, such as an ELF file, bitstream, data files, and so forth. Each image can have multiple partitions, for example an ELF can have multiple loadable sections, each of which form a partition in the boot image. The table also contains the information of number of partitions related to an image. The following table provides the address offsets, parameters, and descriptions for the AMD Zynq™ UltraScale+™ MPSoC.

Table 22: Zynq UltraScale+ MPSoC Device Image Header

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

Zynq UltraScale+ MPSoC Partition Header

About the Partition Header

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

Table 23: Zynq UltraScale+ MPSoC Device Partition Header

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

Table 23: Zynq UltraScale+ MPSoC Device Partition Header (cont'd)

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

Zynq UltraScale+ MPSoC Partition Attribute Bits

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

Table 24: Zynq UltraScale+ MPSoC Device Partition Attribute Bits

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

Table 24: Zynq UltraScale+ MPSoC Device Partition Attribute Bits (cont'd)

Bit Offset	Field Name	Description
18	Endianness	0: Little Endian 1: Big Endian
17:16	Partition Owner	0: FSBL 1: U-Boot 2 and 3: Reserved
15	RSA Authentication Certificate present	0: No RSA Authentication Certificate 1: RSA Authentication Certificate
14:12	Checksum Type	0: None 1-2: Reserved 3: SHA3 4-7: Reserved
11:8	Destination CPU	0: None 1: A53-0 2: A53-1 3: A53-2 4: A53-3 5: R5-0 6: R5 -1 7 R5-lockstep 8: PMU 9-15: Reserved
7	Encryption Present	0: Not Encrypted 1: Encrypted
6:4	Destination Device	0: None 1: PS 2: PL 3-15: Reserved
3	A5X Exec State	0: AARCH64 (default) 1: AARCH32
2:1	Exception Level	0: EL0 1: EL1 2: EL2 3: EL3

Table 24: Zynq UltraScale+ MPSoC Device Partition Attribute Bits (cont'd)

Bit Offset	Field Name	Description
0	Trustzone	0: Non-secure 1: Secure

Zynq UltraScale+ MPSoC Authentication Certificates

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

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

Table 25: Zynq UltraScale+ MPSoC Device Authentication Certificates

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

Note: FSBL Signature is calculated as follows:

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

Zynq UltraScale+ MPSoC Authentication Certification Header

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

Table 26: Authentication Header Bit Fields

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

Zynq UltraScale+ MPSoC Secure Header

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

Figure 43: Zynq UltraScale+ MPSoC Secure Header

AES

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

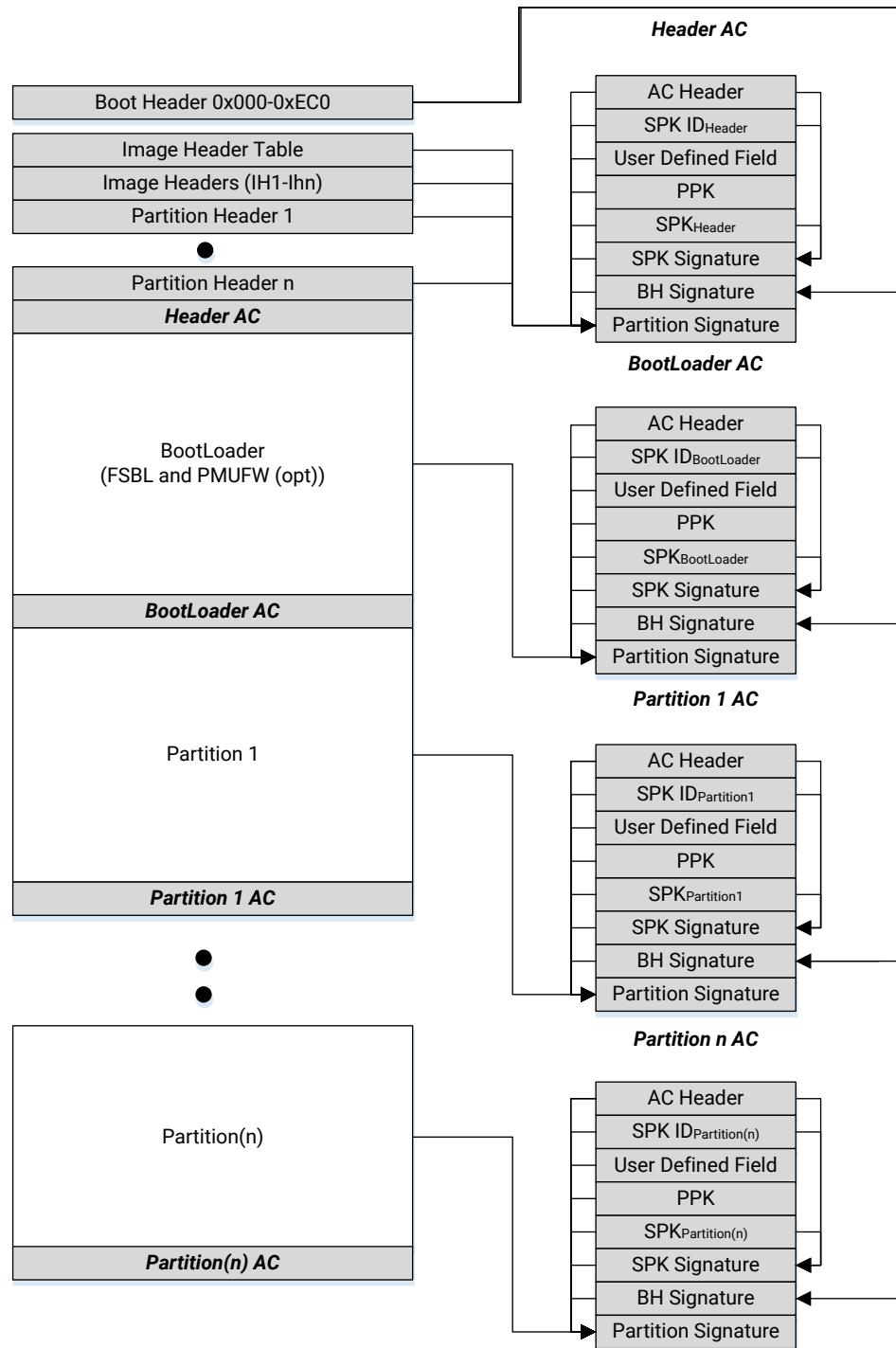
AES with Key rolling

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

Zynq UltraScale+ MPSoC Boot Image Block Diagram

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

Figure 44: Zynq UltraScale+ MPSoC Device Boot Image Block Diagram



X18916-081518

Versal Adaptive SoC Boot Image Format

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

Platform Management Controller

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

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

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

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

Figure 45: Versal Adaptive SoC Boot Image Block Diagram

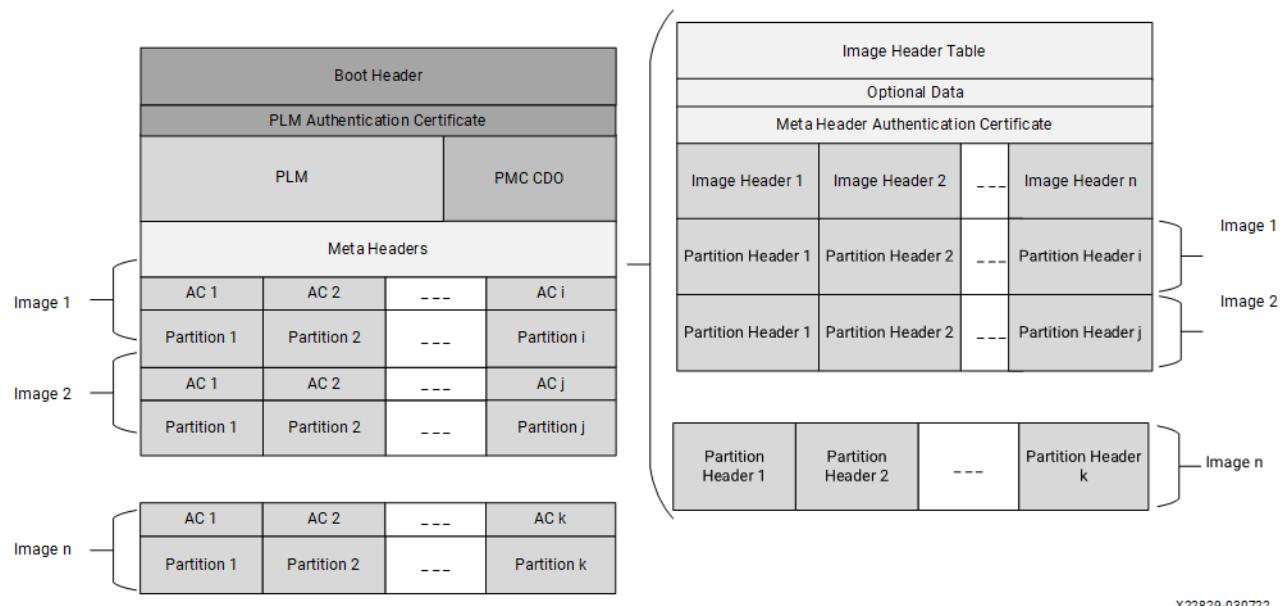
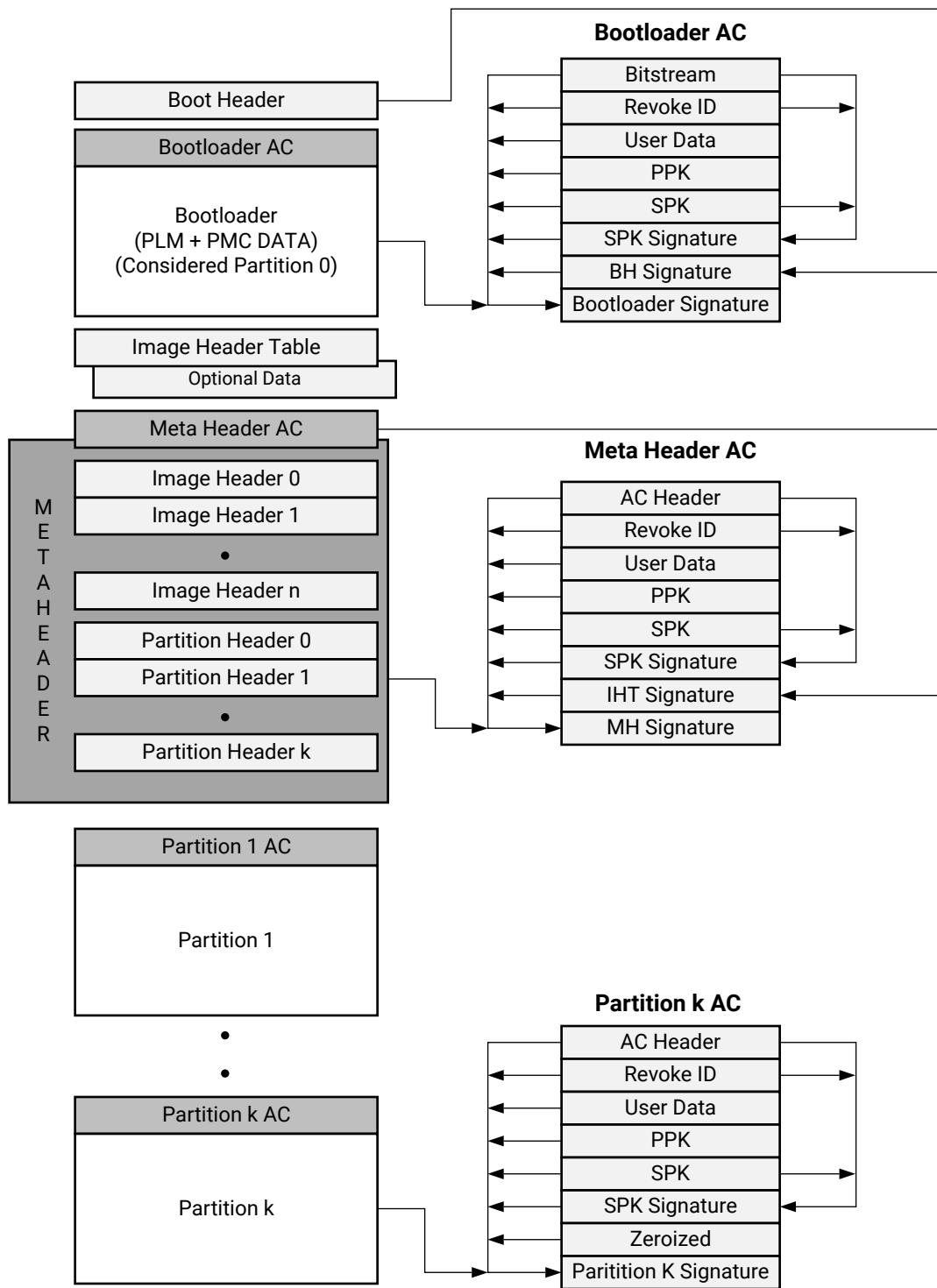


Figure 46: Versal Adaptive SoC Boot Image Block Diagram Part II



X28584-090823

Versal Adaptive SoC Boot Header

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

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

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

Note: The default SelectMAP width is X32.

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

Table 27: Versal Adaptive SoC Boot Header Format

Offset (Hex)	Size (Bytes)	Description	Details
0x00	16	SelectMAP bus width	Used to determine if the SelectMAP bus width is x8, x16, or x32
0x10	4	QSPI bus width	QSPI bus width description. This is required to identify the QSPI flash in single/dual stacked or dual parallel mode. 0xAA995566 in the little endian format.
0x14	4	Image identification	Boot image identification string. Contains 4 bytes X, N, L, X in byte order, which is 0x584c4e58 in the little endian format.
0x18	4	Encryption key source	This field is used to identify the AES key source: 0x00000000 - Unencrypted 0xA5C3C5A3 - eFUSE red key 0xA5C3C5A5 - eFUSE black key 0x3A5C3C5A - BBRAM red key 0x3A5C3C59 - BBRAM black key 0xA35C7C53 - Boot Header black key
0x1C	4	PLM source offset	PLM source start address in PDI
0x20	4	PMC data load address	PMC CDO address to load
0x24	4	PMC data length	PMC CDO length

Table 27: Versal Adaptive SoC Boot Header Format (cont'd)

Offset (Hex)	Size (Bytes)	Description	Details
0x28	4	Total PMC data length	PMC CDO length including authentication and encryption overhead
0x2C	4	PLM length	PLM original image size
0x30	4	Total PLM length	PLM image size including the authentication and encryption overhead
0x34	4	Boot header attributes	Versal Adaptive SoC Boot Header Attributes
0x38	32	Black key	256-bit key, only valid when encryption status is set to black key in boot header
0x58	12	Black IV	Initialization vector used when decrypting the black key
0x64	12	Secure header IV	Secure header initialization vector
0x70	4	PUF shutter value	Length of the time the PUF samples before it closes the shutter Note: This shutter value must match the shutter value that was used during PUF registration.
0x74	12	Secure Header IV for PMC Data	The IV used to decrypt secure header of PMC data.
0x80	68	Reserved	Populate with zeroes.
0xC4	4	Meta Header Offset	Offset to the start of meta header.
0xC8-0x124	96	Reserved	
0x128	2048	Register init	Stores register write pairs for system register initialization
0x928	1544	PUF helper data	PUF helper data
0xF30	4	Checksum	Header checksum
0xF34	76	SHA3 padding	SHA3 standard padding

Versal Adaptive SoC Boot Header Attributes

The image attributes are described in the following table.

Table 28: Versal Adaptive SoC Boot Header Attributes

Field Name	Bit Offset	Width	Default Value	Description
Reserved	[31:18]	14	0x0	Reserved for future use, Must be 0
PUF Mode	[17:16]	2	0x0	0x3 - PUF 4K mode.

Table 28: Versal Adaptive SoC Boot Header Attributes (cont'd)

Field Name	Bit Offset	Width	Default Value	Description
Boot Header Authentication	[15:14]	2	0x0	0x3 - Authentication of the boot image is done, excluding verification of PPK hash and SPK ID. All others - Authentication is decided based on eFUSE RSA/ECDSA bits.
Reserved	[13:12]	2	0x0	Reserved for future use, Must be 0
DPA counter measure	[11:10]	2	0x0	0x3 - Enabled All others disabled. (eFUSE over rides this)
Checksum selection	[9:8]	2	0x0	0x0, 0x1, 0x2 - Reserved 0x3 - SHA3 is used as hash function to do Checksum.
PUF HD	[7:6]	2	0x0	0x3 - PUF HD is part of boot header All other - PUF HD is in eFUSE.
Reserved	[5:0]	6	0x0	Reserved

Versal Adaptive SoC Image Header Table

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

Table 29: Versal Adaptive SoC Image Header Table

Offset	Name	Description
0x0	Version	0x00040000(v4.0) : 1. Added AAD support for IHT. 2. Hash is included into the 32k secure chunk. 0x00030000(v3.0): updated secure chunk size to 32 KB from 64 KB 0x00020000(v2.00): IHT, PHT sizes doubled
0x4	Total Number of Images	Total number of images in the PDI
0x8	Image Header Offset	Word address to start of first image header
0xC	Total Number of Partitions	Total number of partitions in the PDI
0x10	Partition Header Offset	Word offset to the start of partitions headers

Table 29: Versal Adaptive SoC Image Header Table (cont'd)

Offset	Name	Description
0x14	Secondary boot device address	Indicates the address where secondary image is present. This is only valid if secondary boot device is present in attributes
0x1C	Image Header Table Attributes	Refer to Table 30: Versal Adaptive SoC Image Header Table Attributes
0x20	PDI ID	Used to identify a PDI
0x24	Parent ID	ID of initial boot PDI. For boot PDI, it is same as the PDI ID
0x28	Identification string	Full PDI if present with boot header – “FPDI” Partial/Sub-system PDI – “PPDI”
0x2C	Headers size	0-7: Image header table size in words 8-15: Image header size in words 16-23: Partition header size in words 24-31: Reserved
0x30	Total meta header length (Word)	Including authentication and encryption overhead (excluding IHT and including AC)
0x34 -0x3C	IV for encryption of meta header	IV for decrypting SH of header table
0x40	Encryption status	Encryption key source, only key source used for PLM is valid for meta header. 0x00000000 - Unencrypted 0xA5C3C5A3 - eFuse red key 0xA5C3C5A5 - eFUSE black key 0x3A5C3C5A - BBRAM red key 0x3A5C3C59 - BBRAM black key 0xA35C7C53 - Boot Header black key
0x48	Meta Header AC Offset (Word)	Word Offset to Meta Header Authentication Certificate
0x4C	Meta Header Black/IV	IV that is used to encrypt the Black key used to encrypt the Meta Header.
0x58	Optional Data Length (Word)	Size of Optional Data available in Bootloader
0x5C - 0x78	Reserved	0x0
0x7C	Checksum	A sum of all the previous words in the image header table

Image Header Table Attributes

The image header tables are described in the following table.

Table 30: Versal Adaptive SoC Image Header Table Attributes

Bit Field	Name	Description
31:14	Reserved	0

Table 30: Versal Adaptive SoC Image Header Table Attributes (cont'd)

Bit Field	Name	Description
14	PUF Helper Data Location	Location of the PUF Helper Data efuse/BH
12	dpacm enable	DPA Counter Measure enable or not
11:6	Secondary boot device	<p>Indicates the device on which rest of the data is present in.</p> <p>0 - Same boot device (default) 1 - QSPI32 2 - QSPI24 3 - NAND 4 - SD0 5 - SD1 6 - SDLS 7 - MMC 8 - USB 9 - ETHERNET 10 - PCIe 11 - SATA 12 - OSPI 13 - SMAP 14 - SBI 15 - SD0RAW 16 - SD1RAW 17 - SDLSRAW 18 - MMCRAW 19 - MMC0 20 - MMC0RAW 21 - imagestore All others are reserved</p> <p>Note: These options are supported for various devices in Bootgen. For the exact list of secondary boot devices supported by any device, refer to its corresponding Systems Software Developers Guide (SSDG).</p>
5:0		Reserved

IHT Optional Data

Optional Data is binary data placed in the PDI after Image Header Table (IHT). You can include your own optional data. This data is authenticated as part IHT Signature and IHT is added as AAD (Additional Authenticated Data) for the first Secure Header (SH) of the Image Header (IH) during encryption along with IHT.

As part of Authentication Optimization, for partitions that use the same authentication keys, the partition hashes are placed as part of IHT Optional Data. See [Authentication Optimization](#).

Bootgen also accepts user optional data. There can also be multiple entries of optional data. For adding user optional data refer to [optionaldata](#).

Table 31: Versal Adaptive SoC IHT Optional Data

Offset	Name	Description
0x0	ID	The Data IDs from 0x0 to 0x20 for Optional Data are reserved for internal use. User Optional Data ID can be anything > 0x20. 0 - None, can be used for padding 1 - PLM Build Time Configuration Metadata as defined in the PLM Configuration document 2 - Data structure version information used in In-Place PLM Update Compatibility Check 3 - Hash Table for authentication optimization
0x2	Size	Total optional data size in words
0x4	Data	Actual data including padding
Last	Checksum	Sum of all the previous words in this data structure

Versal Adaptive SoC Image Header

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

Table 32: Versal Adaptive SoC Image Header

Offset	Name	Description
0x0	First Partition Header (Word)	Word offset to first partition header
0x4	Number of Partitions	Number of partitions present for this image
0x8	Revoke ID	Revoke ID for Meta Header
0xC	Image Attributes	See Image Attributes table
0x10-0x1C	Image Name	ASCII name of the image. Max of 16 characters. Fill with Zeros when padding is required.
0x20	Image/Node ID	Defines the resource node the image is initializing
0x24	Unique ID	Defines the affinity/compatibility identifier when required for a given device resource

Table 32: Versal Adaptive SoC Image Header (cont'd)

Offset	Name	Description
0x28	Parent Unique ID	Defines the required parent resource UID for the configuration content of the image, if required
0x2c	Function ID	Identifier used to capture the unique function of the image configuration data
0x30	DDR Low Address for Image Copy	The DDR lower 32-bit address where the image must be copied when memcpy is enabled in BIF
0.34	DDR High Address for Image Copy	The DDR higher 32-bit address where image must be copied when memcpy is enabled in BIF
0x38	Reserved	
0x3C	Checksum	A sum of all the previous words.

The following table shows the Image Header Attributes.

Table 33: Versal Adaptive SoC Image Header Attributes

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

Versal Adaptive SoC Partition Header

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

Table 34: Versal Adaptive SoC Partition Header Table

Offset	Name	Description
0x0	Partition Data Word Length	Encrypted partition data length
0x4	Extracted Data Word Length	Unencrypted data length
0x8	Total Partition Word Length (Includes Authentication Certificate)	The total encrypted + padding + expansion + authentication length
0xC	Next Partition header offset (Word)	Offset of next partition header

Table 34: Versal Adaptive SoC Partition Header Table (cont'd)

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

Table 34: Versal Adaptive SoC Partition Header Table (cont'd)

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

The following table lists the partition header table attributes.

Table 35: Versal Adaptive SoC Partition Header Table Attributes

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

Table 35: Versal Adaptive SoC Partition Header Table Attributes (cont'd)

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

Versal Adaptive SoC Authentication Certificates

The Authentication Certificate is a structure that contains all the information related to the authentication of a partition. This structure has the public keys and the signatures that BootROM/PLM needs to verify. There is an Authentication Header in each Authentication Certificate, which gives information like the key sizes, algorithm used for signing, and so forth. Unlike the other devices, the Authentication Certificate is prepended or attached to the beginning of the actual partition, for which authentication is enabled. If you want Bootgen to perform authentication on the meta headers, specify it explicitly under the 'metaheader' bif attribute. See [BIF Attribute Reference](#) for information on usage.

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

Table 36: Versal Adaptive SoC Authentication Certificate – ECDSA p384

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

Table 37: Versal Adaptive SoC Authentication Certificate – ECDSA p521

Authentication Certificate Bits		Description
0x00		Authentication Header. See Versal Adaptive SoC Authentication Certification Header
0x04		Revoke ID
0x08		UDF (56 bytes)

Table 37: Versal Adaptive SoC Authentication Certificate - ECDSA p521 (cont'd)

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

Table 38: Versal Adaptive SoC Authentication Certificate - RSA

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

Versal Adaptive SoC Authentication Certification Header

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

Table 39: Authentication Header Bit Fields

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

Notes:

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

Creating Boot Images

Boot Image Format (BIF)

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

- Configuration attributes to create secure/non-secure boot images
- Bootloader
 - First stage bootloader (FSBL) for AMD Zynq™ devices and AMD Zynq™ UltraScale+™ MPSoCs
 - Platform loader and manager (PLM) for AMD Versal™ adaptive SoC
 - **Note:** It is recommended to use the same release version of bootloader (FSBL/PLM) and Bootgen together.
- One or more partition images

Along with properties and attributes, Bootgen takes multiple commands to define the behavior while it is creating the boot images. For example, to create a boot image for a qualified FPGA device, an AMD Zynq™ 7000 SoC device, AMD Versal™ adaptive SoC series, or an AMD Zynq™ UltraScale+™ MPSoC device, you must provide the appropriate [arch](#) command option to Bootgen. The following appendices list and describe the available options to direct Bootgen behavior.

- [Use Cases and Examples](#)
- [BIF Attribute Reference](#)
- [Command Reference](#)

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

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

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

Note: Refer to *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)* for more information

BIF Syntax and Supported File Types

The BIF file specifies each component of the boot image, in order of boot, and allows optional attributes to be applied to each image component. In some cases, an image component can be mapped to more than one partition if the image component is not contiguous in memory. For example, if an ELF file has multiple loadable sections that are non-contiguous, then each section can be a separate partition. BIF file syntax takes the following form:

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

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

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

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

The following example is of a BIF with additional white space and new lines for improved readability:

```
<bootimage_name>:  
{  
    /* common attributes */  
    [attribute1] <argument1>  
  
    /* bootloader */  
    [attribute2,  
     attribute3,  
     attribute4=<argument>  
    ] <elf>  
  
    /* pl bitstream */  
    [  
        attribute2,  
        attribute3,  
        attribute4=<argument>,  
        attribute=<argument>  
    ] <bit>  
  
    /* another elf partition */  
    [  
        attribute3  
    ] <elf>  
  
    /* bin partition */  
    <bin>  
}
```

Bootgen Supported Files

The following table lists the Bootgen supported files.

Table 40: Bootgen Supported Files

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

Table 40: Bootgen Supported Files (cont'd)

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

BIF Syntax for Versal Adaptive SoC

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

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

```

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

```

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

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

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

Attributes

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

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

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

Table 41: Bootgen Attributes and Description

Option/Attribute	Description	Used By
<code>aarch32_mode</code>	Specifies the binary file that is to be executed in 32-bit mode.	<ul style="list-style-type: none">• <code>zynqmp</code>• <code>versal</code>
<code>aeskeyfile <aes_key_filepath></code>	The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file needs to be written to eFUSE or BBRAM. If the key file is not present in the path specified, a new key is generated by Bootgen, which is used for encryption. For example: If encryption is selected for bitstream in the <code>BIF</code> file, the output is an encrypted bitstream.	<ul style="list-style-type: none">• All
<code>alignment <byte></code>	Sets the byte alignment. The partition is padded to be aligned to a multiple of this value. This attribute cannot be used with offset.	<ul style="list-style-type: none">• <code>zynq</code>• <code>zynqmp</code>
<code>auth_params <options></code>	Extra options for authentication: <ul style="list-style-type: none">• <code>ppk_select</code>: 0=1, 1=2 of two PPKs supported.• <code>spk_id</code>: 32-bit ID to differentiate SPKs.• <code>spk_select</code>: To differentiate spk and user eFUSES. Default is spk-efuse.• <code>header_auth</code>: To authenticate headers when no partition is authenticated.	<ul style="list-style-type: none">• <code>zynqmp</code>
<code>authentication <option></code>	Specifies the partition to be authenticated. <ul style="list-style-type: none">• Authentication for Zynq is done using RSA-2048.• Authentication for Zynq UltraScale+ MPSoCs is done using RSA-4096.• Authentication for Versal adaptive SoC is done using RSA-4096, ECDSA-p384, and ECDSA-p521. The arguments are: <ul style="list-style-type: none">• <code>none</code>: Partition not signed.• <code>ecdsa-p384</code>: partition signed using ecdsa-p384 curve• <code>ecdsa-p521</code>: partition signed using ecdsa-p521 curve• <code>rsa</code>: Partition signed using RSA algorithm.	<ul style="list-style-type: none">• All

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>bbram_kek_iv <filename></code>	Specifies the IV that is used to encrypt the corresponding key. <code>bbram_kek_iv</code> is valid with <code>keysrc=bbram_blk_key</code> .	• versal
<code>bh_kek_iv <filename></code>	Specifies the IV that is used to encrypt the corresponding key. <code>bh_kek_iv</code> is valid with <code>keysrc=bh_blk_key</code> .	• versal
<code>bh_key_iv <filename></code>	Initialization vector used when decrypting the obfuscated key or a black key.	• zynqmp
<code>bh_keyfile <filename></code>	256-bit obfuscated key or black key to be stored in the Boot Header. This is only valid when <code>keysrc</code> for encryption is <code>bh_gry_key</code> or <code>bh_blk_key</code> . Note: Obfuscated key is not supported for Versal devices.	• zynqmp • versal
<code>bhsignature <filename></code>	Imports Boot Header signature into authentication certificate. This can be used if you do not want to share the secret key PSK. You can create a signature and provide it to Bootgen. The file format is <code>bootheader.sha384.sig</code> .	• zynqmp • versal
<code>big_endian</code>	Specifies the binary file is in big endian format.	• zynqmp • versal
<code>blocks <block sizes></code>	Specifies block sizes for key-rolling feature in Encryption. Each module is encrypted using its own unique key. The initial key is stored at the key source on the device, while keys for each successive blocks are encrypted (wrapped) in the previous module.	• zynqmp • versal
<code>boot_config <options></code>	This attribute specifies the parameters that are used to configure the boot image.	• versal

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>boot_device <options></code>	<p>Specifies the secondary boot device. Indicates the device on which the partition is present. Options are:</p> <ul style="list-style-type: none"> • qspi32 • qspi24 • nand • sd0 • sd1 • sd-1s • mmc • usb • ethernet • pcie • sata • ospi • smap • sbi • sd0-raw • sd1-raw • sd-1s-raw • mmc-raw • mmc0 • mmc0-raw <p>Note: These options are supported for various devices in Bootgen. For a list of secondary boot options, see the <i>Versal Adaptive SoC System Software Developers Guide</i> (UG1304) or the <i>Zynq UltraScale+ MPSoC: Software Developers Guide</i> (UG1137). For hardware/register/interface information and primary boot modes, refer to the corresponding TRM, such as the <i>Zynq UltraScale+ Device Technical Reference Manual</i> (UG1085), the <i>Versal Adaptive SoC Technical Reference Manual</i> (AM011), or the <i>Versal Adaptive SoC Register Reference</i> (AM012).</p>	<ul style="list-style-type: none"> • zynqmp • versal
<code>bootimage <filename.bin></code>	Specifies that the listed input file is a boot image that was created by Bootgen.	<ul style="list-style-type: none"> • zynq • zynqmp • versal
<code>bootloader <partition></code>	Specifies the partition is a bootloader (FSBL/PLM). This attribute is specified along with other partition BIF attributes.	<ul style="list-style-type: none"> • zynq • zynqmp • versal
<code>bootvectors <vector_values></code>	Specifies the vector table for execute in place (XIP).	<ul style="list-style-type: none"> • zynqmp

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>checksum <options></code>	<p>Specifies that the partition needs to be checksummed. This option is not supported along with more secure features like authentication and encryption. Checksum algorithms are:</p> <ul style="list-style-type: none"> <code>none</code>: No checksum operation. <code>md5</code>: For AMD Zynq™ 7000 SoC devices only <code>sha3</code>: For AMD Zynq™ UltraScale+™ MPSoC and Versal devices. <p>Note: Zynq devices do not support checksum for bootloaders. The following devices do support checksum operation for bootloaders:</p> <ul style="list-style-type: none"> Zynq UltraScale+ MPSoC Versal adaptive SoC 	<ul style="list-style-type: none"> <code>zynq</code> <code>zynqmp</code> <code>versal</code>
<code>copy <address></code>	This attribute specifies that the image is to be copied to memory at specified address.	<ul style="list-style-type: none"> <code>versal</code>
<code>core <options></code>	<p>This attribute specifies which core executes the partition. The options for AMD Versal™ adaptive SoC are:</p> <ul style="list-style-type: none"> <code>a72-0</code> <code>a72-1</code> <code>r5-0</code> <code>r5-1</code> <code>psm</code> <code>aie</code> <code>r5-lockstep</code> 	<ul style="list-style-type: none"> <code>versal</code>
<code>delay_handoff</code>	This attribute specifies that the hand-off to the subsystem/image is delayed.	<ul style="list-style-type: none"> <code>versal</code>
<code>delay_load</code>	This attribute specifies that the loading of the subsystem/image is delayed.	<ul style="list-style-type: none"> <code>versal</code>
<code>delay_auth</code>	Indicates that the authentication is done at a later stage. This helps bootgen to reserve space for hashes during partition encryption.	<ul style="list-style-type: none"> <code>versal</code>
<code>destination_device <device_type></code>	<p>This specifies if the partition is targeted for PS or PL. The options are:</p> <ul style="list-style-type: none"> <code>ps</code>: the partition is targeted for PS (default). <code>pl</code>: the partition is targeted for PL, for bitstreams. 	<ul style="list-style-type: none"> <code>zynqmp</code>
<code>destination_cpu <device_core></code>	<p>Specifies the core on which the partition should be executed.</p> <ul style="list-style-type: none"> <code>a53-0</code> <code>a53-1</code> <code>a53-2</code> <code>a53-3</code> <code>r5-0</code> (default) <code>r5-1</code> <code>pmu</code> <code>r5-lockstep</code> 	<ul style="list-style-type: none"> <code>zynqmp</code>

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>early_handoff</code>	This flag ensures that the handoff to applications that are critical immediately after the partition is loaded; otherwise, all the partitions are loaded sequentially first, and then the handoff also happens in a sequential fashion.	• zynqmp
<code>efuse_kek_iv <filename></code>	Specifies the IV that is used to encrypt the corresponding key. <code>efuse_kek_iv</code> is valid with <code>keysrc=efuse_blk_key</code> .	• versal
<code>efuse_user_kek0_iv <filename></code>	Specifies the IV that is used to encrypt the corresponding key. <code>efuse_user_kek0_iv</code> is valid with <code>keysrc=efuse_user_blk_key0</code> .	• versal
<code>efuse_user_kek1_iv <filename></code>	Specifies the IV that is used to encrypt the corresponding key. <code>efuse_user_kek1_iv</code> is valid with <code>keysrc=efuse_user_blk_key1</code> .	• versal
<code>encryption <option></code>	Specifies the partition to be encrypted. Encryption algorithms are: zynq uses AES-CBC, while zynqmp and Versal use AES-GCM. The partition options are: <ul style="list-style-type: none">• none: Partition not encrypted.• aes: Partition encrypted using AES algorithm.	• All
<code>exception_level <options></code>	Exception level for which the core should be configured. Options are: <ul style="list-style-type: none">• el-0• el-1• el-2• el-3	• zynqmp • versal
<code>familykey <key file></code>	Specifies the family key.	• zynqmp • fpga
<code>file <path/to/file></code>	This attribute specifies the file for creating the partition.	• versal

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>fsbl_config <options></code>	<p>Specifies the sub-attributes used to configure the bootimage. Those sub-attributes are:</p> <ul style="list-style-type: none"> <code>bh_auth_enable</code>: RSA authentication of the boot image is done excluding the verification of PPK hash and SPK ID. <code>auth_only</code>: boot image is only RSA signed. FSBL should not be decrypted. <code>opt_key</code>: Operational key is used for block-0 decryption. Secure Header has the opt key. <code>pufhd_bh</code>: PUF helper data is stored in Boot Header (Default is <code>efuse</code>). PUF helper data file is passed to Bootgen using the <code>[puf_file]</code> option. <code>puf4kmode</code>: PUF is tuned to use in 4k bit configuration. <code>shutter = <value></code>: 32 bit <code>PUF_SHUT</code> register value to configure PUF for shutter offset time and shutter open time. <p>Note: This shutter value must match the shutter value that was used during PUF registration.</p>	<ul style="list-style-type: none"> <code>zynqmp</code>
<code>headersignature <signature_file></code>	Imports the header signature into an Authentication Certificate. This can be used in case the user does not want to share the secret key. The user can create a signature and provide it to Bootgen.	<ul style="list-style-type: none"> <code>zynq</code> <code>zynqmp</code> <code>versal</code>
<code>hivec</code>	<p>Specifies the location of exception vector table as <code>hivec</code> (Hi-Vector). The default value is <code>lovec</code> (Low-Vector). This is applicable with A53 (32 bit) and R5 cores only.</p> <ul style="list-style-type: none"> <code>hivec</code>: exception vector table at <code>0xFFFF0000</code>. <code>lovec</code>: exception vector table at <code>0x00000000</code>. 	<ul style="list-style-type: none"> <code>zynqmp</code>
<code>id <id></code>	This attribute specifies the following IDs based on the place its defined: <ul style="list-style-type: none"> <code>pdi id</code> - within outermost/PDI parenthesis <code>image id</code> - within image parenthesis <code>partition id</code> - within partition parenthesis 	<ul style="list-style-type: none"> <code>versal</code>
<code>image</code>	Defines a subsystem/image.	<ul style="list-style-type: none"> <code>versal</code>
<code>init <filename></code>	Register initialization block at the end of the bootloader, built by parsing the init (.int) file specification. A maximum of 256 address-value init pairs are allowed. The init files have a specific format.	<ul style="list-style-type: none"> <code>zynq</code> <code>zynqmp</code> <code>versal</code>

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>keysrc</code>	Specifies key source for encryption for Versal adaptive SoC. The keysrc can be specified for individual partitions. <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> 	<ul style="list-style-type: none"> versal
<code>keysrc_encryption</code>	Specifies the key source for encryption. The keys are: <ul style="list-style-type: none"> <code>efuse_gry_key</code>: Grey (Obfuscated) Key stored in eFUSE. See Gray/Obfuscated Keys <code>bh_gry_key</code>: Grey (Obfuscated) Key stored in boot header. <code>bh_blk_key</code>: Black Key stored in boot header. See Black/PUF Keys <code>efuse_blk_key</code>: Black Key stored in eFUSE. <code>kup_key</code>: User Key. <code>efuse_red_key</code>: Red key stored in eFUSE. See Rolling Keys. <code>bbram_red_key</code>: Red key stored in BBRAM. 	<ul style="list-style-type: none"> zynq zynqmp
<code>load <address></code>	Sets the desired load address for the partition in memory.	<ul style="list-style-type: none"> zynq zynqmp versal
<code>metaheader</code>	This attribute is used to define encryption and authentication attributes for meta headers like keys, key sources, and so on.	<ul style="list-style-type: none"> versal
<code>name <name></code>	This attribute specifies the name of the image/subsystem.	<ul style="list-style-type: none"> versal
<code>offset <offset></code>	Sets the absolute offset of the partition in the boot image.	<ul style="list-style-type: none"> zynq zynqmp versal

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>optionaldata {<filename>, id=<id>}</code>	This allows you to specify data ID and data file.	<ul style="list-style-type: none"> versal
<code>parent_id</code>	This attribute specifies the ID for the parent PDI. This is used to identify the relationship between a partial PDI and its corresponding boot PDI.	<ul style="list-style-type: none"> versal
<code>partition</code>	This attribute is used to define a partition. It is an optional attribute to make the BIF short and readable.	<ul style="list-style-type: none"> versal
<code>partition_owner, owner <option></code>	<p>Owner of the partition which is responsible to load the partition. Options are:</p> <p>For Zynq/Zynq UltraScale+ MPSoC:</p> <ul style="list-style-type: none"> <code>fsbl</code>: Partition is loaded by FSBL. <code>uboot</code>: Partition is loaded by U-Boot. <p>For Versal:</p> <ul style="list-style-type: none"> <code>plm</code>: partition loaded by PLM. <code>non-plm</code>: partition is not loaded by PLM, but it is loaded by another entity like U-Boot. 	<ul style="list-style-type: none"> zynq zynqmp versal
<code>pid <ID></code>	Specifies the Partition ID. PID can be a 32-bit value (0 to 0xFFFFFFFF).	<ul style="list-style-type: none"> zynqmp
<code>pmufw_image <image_name></code>	PMU firmware image to be loaded by BootROM, before loading the FSBL.	<ul style="list-style-type: none"> zynqmp
<code>ppkfile <key filename></code>	Primary Public Key (PPK). Used to authenticate partitions in the boot image. See Using Authentication for more information.	<ul style="list-style-type: none"> zynq zynqmp versal
<code>presign <sig_filename></code>	Partition signature (<code>.sig</code>) file.	<ul style="list-style-type: none"> zynq zynqmp fpga
<code>pskfile <key filename></code>	Primary Secret Key (PSK). Used to authenticate partitions in the boot image. See the Using Authentication for more information.	<ul style="list-style-type: none"> zynq zynqmp versal
<code>puf_file <filename></code>	PUF helper data file. PUF is used with black key as encryption key source. PUF helper data is of 1544 bytes.1536 bytes of PUF HD + 4 bytes of HASH + 3 bytes of AUX + 1 byte alignment.	<ul style="list-style-type: none"> zynqmp versal
<code>reserve <size in bytes></code>	Reserves the memory, which is padded after the partition.	<ul style="list-style-type: none"> zynq zynqmp versal
<code>spk_select <SPK_ID></code>	Specify an SPK ID in user eFUSE.	<ul style="list-style-type: none"> zynqmp
<code>spkfile <filename></code>	Keys used to authenticate partitions in the boot image. See Using Authentication for more information.	<ul style="list-style-type: none"> All
<code>spksignature <signature_file></code>	Imports the SPK signature into an Authentication Certificate. See Using Authentication . This can be used in case the user does not want to share the secret key PSK, The user can create a signature and provide it to Bootgen.	<ul style="list-style-type: none"> zynq zynqmp versal

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>split <options></code>	<p>Splits the image into parts, based on the mode. Split options are:</p> <ul style="list-style-type: none"> • Slaveboot: Supported for Zynq UltraScale+ MPSoC only. Splits as follows: • Boot Header + Bootloader • Image and Partition Headers • Rest of the partitions • normal: Supported for zynq, zynqmp, and versal. Splits as follows: • Booheader + Image Headers + Partition Headers + Bootloader • Partition1 • Partition2 and so on <p>Along with the split mode, output format can also be specified as <code>bin</code> or <code>mcs</code>.</p> <p>Note: The option split mode normal is same as the command line option <code>split</code>. This command line option is deprecated. Split ulaveboot is supported only for Zynq UltraScale+ MPSoC.</p>	<ul style="list-style-type: none"> • zynq • zynqmp • versal
<code>sskfile <key filename></code>	Secondary Secret Key (SSK) key authenticates partitions in the Boot Image. The primary keys authenticate the secondary keys; the secondary keys authenticate the partitions.	<ul style="list-style-type: none"> • All
<code>startup <address></code>	Sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute.	<ul style="list-style-type: none"> • zynq • zynqmp • versal
<code>trustzone <option></code>	The trustzone options are:	<ul style="list-style-type: none"> • zynqmp • versal
<code>type <options></code>	This attribute specifies the type of partition. The options are: <ul style="list-style-type: none"> • bootloader • pmcdata • cdo • cfi • cfi-gsc • bootimage • slr-boot • slr-config 	<ul style="list-style-type: none"> • versal
<code>udf_bh <data_file></code>	Imports a file of data to be copied to the user defined field (UDF) of the Boot Header. The UDF is provided through a text file in the form of a hex string. Total number of bytes in UDF are: zynq = 76 bytes; zynqmp = 40 bytes.	<ul style="list-style-type: none"> • zynq • zynqmp
<code>udf_data <data_file></code>	Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate.	<ul style="list-style-type: none"> • zynq • zynqmp

Table 41: Bootgen Attributes and Description (cont'd)

Option/Attribute	Description	Used By
<code>userkeys <filename></code>	The path to the user keyfile.	<ul style="list-style-type: none">versal
<code>xip_mode</code>	Indicates eXecute In Place (XIP) for FSBL to be executed directly from QSPI flash.	<ul style="list-style-type: none">zynqzynqmp

Using Bootgen GUI

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

Launch Bootgen GUI

The Vitis Unified IDE, introduced in 2024.1, uses the [Theia environment](#), however the Vitis Classic IDE uses Eclipse. The Bootgen functions the same in both the Vitis Classic, and Vitis Unified. However, the layout of the GUI is different. In this section, both the GUI flows for Vitis Unified and Vitis Classic are discussed.

To create a boot image using both the AMD Vitis™ Classic and Unified IDE, do either of the following:

- For Vitis Unified IDE, highlight the application component in the Vitis Components view and then in the Flow view, select **Create Boot Image**.
- For Vitis Classic IDE, Select the application project in the Project Navigator or C/C++ Projects view and right-click **Create Boot Image**.
- Alternatively, for both Vitis Classic and Unified IDE click **Vitis** → **Create Boot Image** → **Target Device Family**.

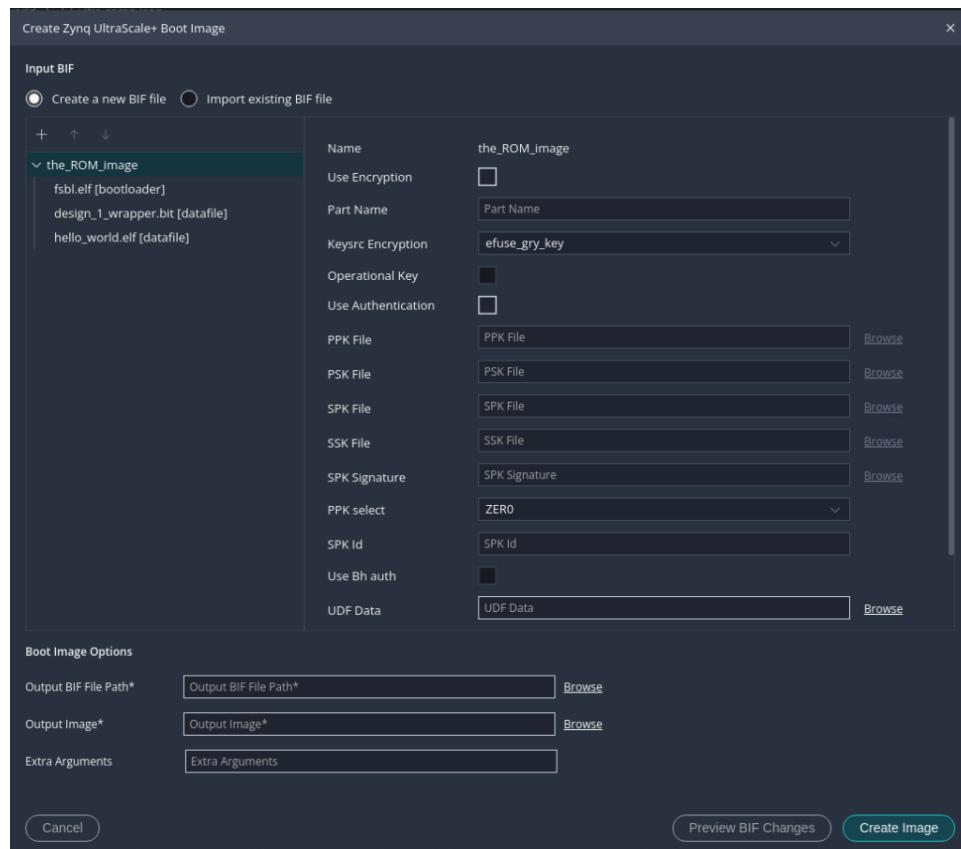
The supported device families are:

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

Bootgen GUI for Zynq 7000 and Zynq UltraScale+ Devices

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

Figure 47: Create Boot Image for Zynq and Zynq UltraScale+ Devices for Vitis Unified IDE



- When you run Create Boot Image the first time for an application, the dialog box is pre-populated with paths to the FSBL ELF file, and the bitstream for the selected hardware (if it exists in hardware project), and then the selected application ELF file.
 - If a boot image was run previously for the application, and a BIF file exists, the dialog box is pre-populated with the values from the /bif folder.
1. Populate the Create Boot Image dialog box with the following information:
 - a. From the **Architecture** drop-down, select the required architecture.
 - b. Select either **Create a BIF file** or **Import an existing BIF file**.

- c. From the Basic tab, specify the **Output BIF file path**.
 - d. If applicable, specify the **UDF data**: See [udf_data](#) for more information about this option.
 - e. Specify the **Output path**.
2. In the Boot image partitions, click the **Add** button to add additional partition images.
 3. Create offset, alignment, and allocation values for partitions in the boot image, if applicable.

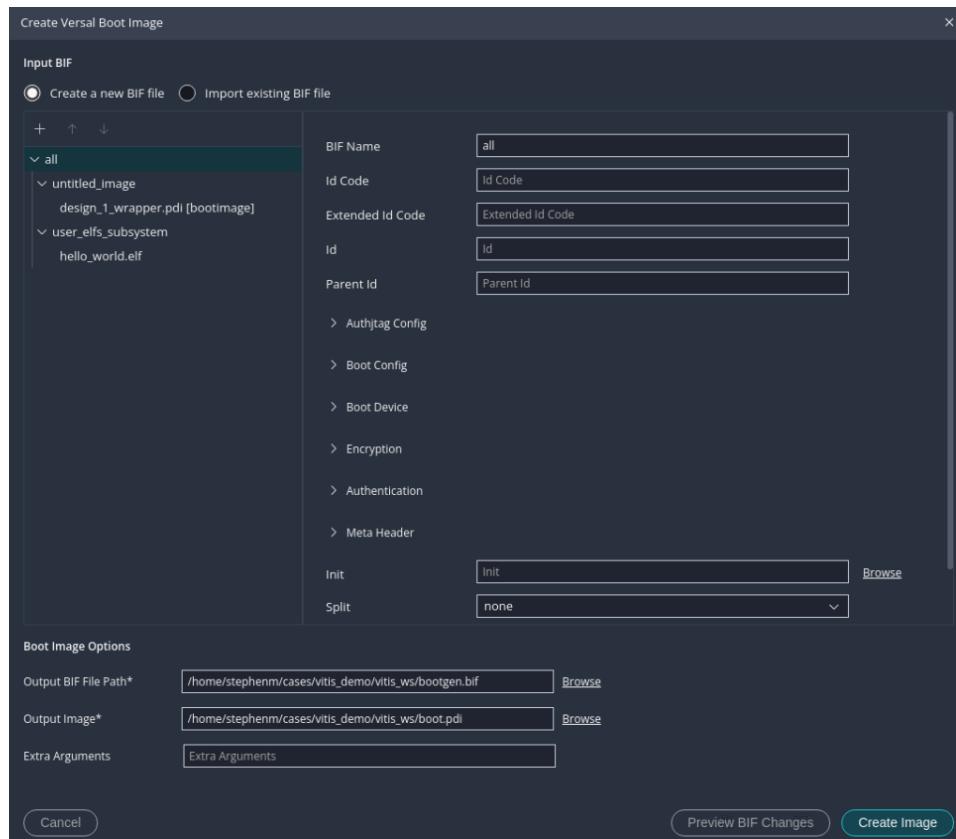
The output file path is set to the `/bif` folder under the selected application project by default.
 4. From the Security tab, you can specify the attributes to create a secure image. This security can be applied to individual partitions as required.
 - a. To enable authentication for a partition, check the **Use Authentication** option, then specify the PPK, SPK, PSK, and SSK values. See the [Using Authentication](#) topic for more information.
 - b. To enable encryption for a partition, select the Encryption view, and check the **Use Encryption** option. See [Using Encryption](#) for more information.
 5. Create or import a BIF file boot image one partition at a time, starting from the bootloader. The partitions list displays the summary of the partitions in the BIF file. It shows the file path, encryption settings, and authentication settings. Use this area to add, delete, modify, and reorder the partitions. You can also set values for enabling encryption, authentication, and checksum, and specifying some other partition related values like **Load**, **Alignment**, and **Offset**

Using Bootgen GUI Options for Versal Adaptive SoCs

Create Versal Boot Image from Flow navigator

If you launch Bootgen GUI for AMD Versal™ from flow navigator, the Create Boot Image dialog box opens, with default values pre-selected from the context of the selected project.

Figure 48: Bootgen GUI for Versal Adaptive SoCs in Vitis Unified IDE



1. Populate the Create Boot Image dialog box with the following information:
 - a. Select either **Create a BIF file** or **Import an existing BIF file**.
Note: AMD Vivado™ generated BIF can be found at <design>.runs/impl_1/ directory.
 - b. From the Basic tab, specify the **Output BIF file path**.
 - c. Specify the **Output Image**.
2. In the Boot image partitions, click the **Add** button to add additional partition images.
3. Create offset, alignment, and allocation values for partitions in the boot image, if applicable.
The output file path is set to the `/bif` folder under the selected application project by default.
4. From the Security tab, you can specify the overall attributes to create a secure image. Partition security attributes can be updated when you select the **partition** and click **Edit** button, then go to Security tab.

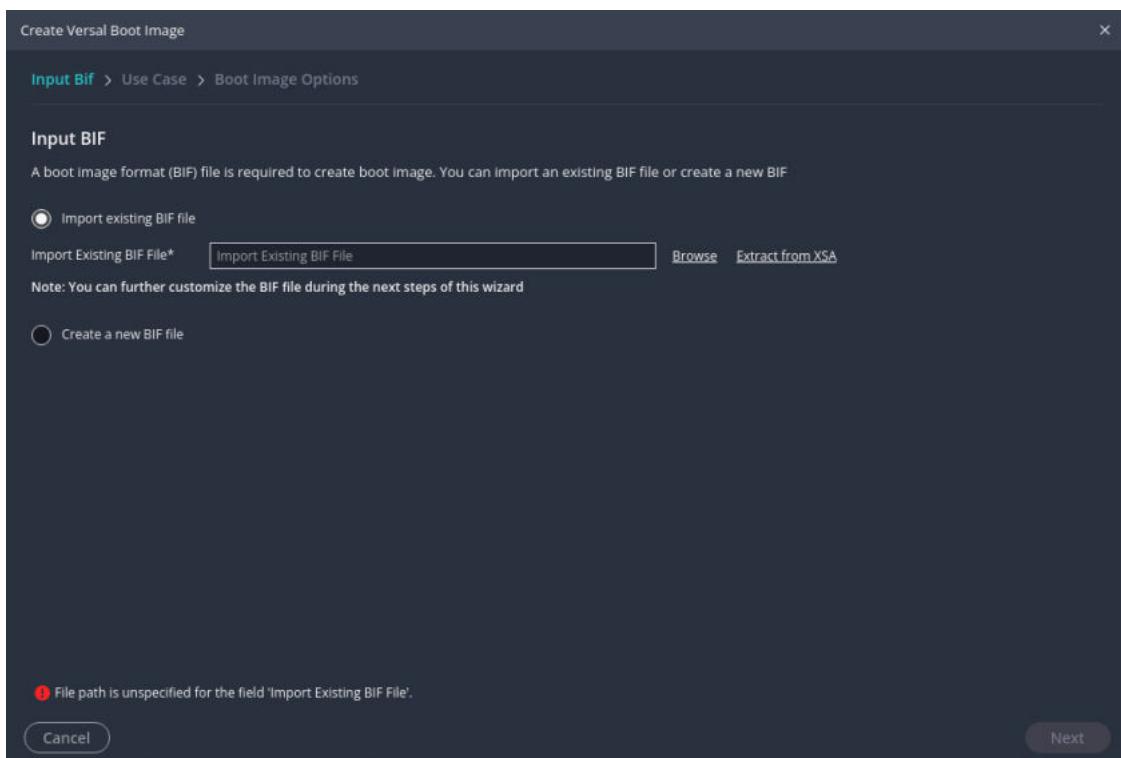
See [Using Authentication](#) and [Using Encryption](#) for more information.

5. Create or import a BIF file boot image one partition at a time, starting from the bootloader. The partitions list displays the summary of the partitions in the BIF file. It shows the file path, encryption settings, and authentication settings. Use this area to add, delete, modify, and reorder the partitions. You can also set values for enabling encryption, authentication, and checksum, and specifying some other partition related values like **Load**, **Alignment**, and **Offset**
6. Contents in Extra Bif attributes dialogue and the **Edit partition → Extra Partition attributes** is appended to the overall BIF file or the partition. You can use these fields to add custom attributes if they are not supported by the Bootgen GUI.

Common Use Cases for Creating Versal Boot Image

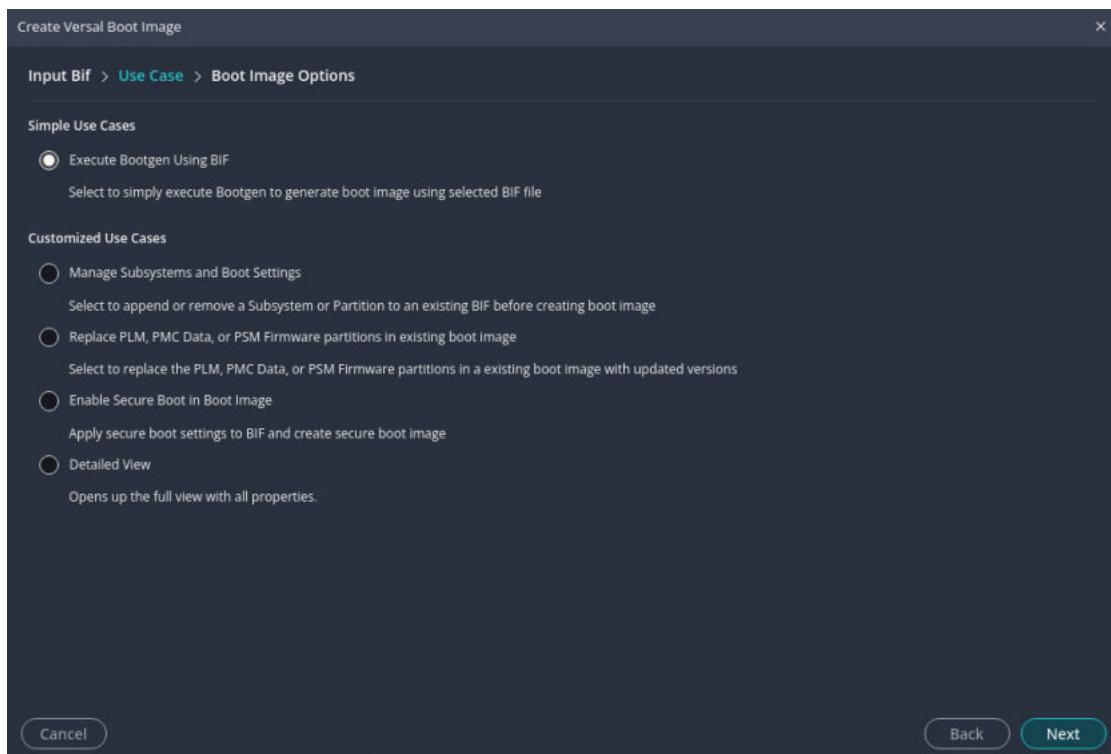
Launching Bootgen GUI for AMD Versal™ from Vitis menu, the Create Boot Image dialog box opens.

Figure 49: Create Boot Image



Select to import existing BIF file, either to use existing BIF file or obtain it by extracting from XSA file. Then the common use case for quickly creating boot image dialog would appear.

Figure 50: Create Versal Boot Image



User could go to corresponding customized use cases and create boot image quickly.

Using Bootgen on the Command Line

When you specify Bootgen options on the command line you have many more options than those provided in the Vitis IDE. In the standard install of the Vitis software platform, the XSCT is available for use as an interactive command line environment, or to use for creating scripting. In the XSCT, you can run Bootgen commands. XSCT accesses the Bootgen executable, which is a separate tool. This Bootgen executable can be installed standalone as described in [Installing Bootgen](#). This is the same tool as is called from the XSCT, hence any scripts developed here or in the XSCT works in the other tool.

The [Software Command-Line Tool](#) in the *Vitis Embedded Software Development Flow Documentation* (UG1400) describes the tool. See the "XSCT Use Cases" chapter for an example of using Bootgen commands in XSCT.

Commands and Descriptions

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

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

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

Table 42: Bootgen Command and Descriptions

Commands	Description and Options	Used by
<code>arch <type></code>	AMD device architecture. Options: <ul style="list-style-type: none">• <code>zynq</code> (default)• <code>zynqmp</code>• <code>fpga</code>• <code>versal</code>	<ul style="list-style-type: none">• All
<code>authenticatedjtag <options></code>	Used to enable JTAG during secure boot. The arguments are: <ul style="list-style-type: none">• <code>rsa</code>• <code>ecdsa</code>	<ul style="list-style-type: none">• <code>versal</code>
<code>bif_help</code>	Prints out the BIF help summary.	<ul style="list-style-type: none">• All
<code>dual_qspi_mode <configuration></code>	Generates two output files for dual QSPI configurations: <ul style="list-style-type: none">• <code>parallel</code>• <code>stacked <size></code>	<ul style="list-style-type: none">• <code>zynq</code>• <code>zynqmp</code>• <code>versal</code>
<code>dual_ospf_mode stacked <size></code>	Generates two output files for stacked configuration.	<ul style="list-style-type: none">• <code>versal</code>

Table 42: Bootgen Command and Descriptions (cont'd)

Commands	Description and Options	Used by
<code>dump <options></code>	<p>Dumps the partition or boot header as per options specified.</p> <ul style="list-style-type: none"> <code>empty</code>: Dumps the partitions as binary files. <code>bh</code>: Dumps boot header as a binary file. <code>plm</code>: Dumps PLM as a binary file. <code>pmc_cdo</code>: Dumps PMC CDO as a binary file. <code>boot_files</code>: Dumps boot header, PLM, and PMC CDO as three separate binary files. <code>slave_pdls</code>: Dumps Slave PDIs for SSI technology use cases <p>The slave PDIs are not dumped by default. This option should be used if you want to debug or analyze the slave PDI separately.</p>	<ul style="list-style-type: none"> versal
<code>dump_dir</code>	Dumps components in specified directory.	<ul style="list-style-type: none"> versal
<code>efuseppkbits <PPK_filename></code>	Generates a PPK hash for eFUSE.	<ul style="list-style-type: none"> zynq zynqmp versal
<code>enable_auth_opt</code>	Used to enable authentication optimization	<ul style="list-style-type: none"> versal
<code>encrypt <options></code>	AES Key storage in device. Options are: <ul style="list-style-type: none"> <code>bbram</code> (default) <code>efuse</code> 	<ul style="list-style-type: none"> zynq fpga
<code>encryption_dump</code>	Generates encryption log file, <code>aes_log.txt</code> .	<ul style="list-style-type: none"> zynqmp versal
<code>fill <hex_byte></code>	Specifies the fill byte to use for padding.	<ul style="list-style-type: none"> zynq zynqmp versal
<code>generate_hashes</code>	Generates file containing padded hash: <ul style="list-style-type: none"> Zynq devices: SHA-2 with PKCS#1v1.5 padding scheme Zynq UltraScale+ MPSoC: SHA-3 with PKCS#1v1.5 padding scheme Versal adaptive SoC: SHA-3 with PSS padding scheme 	<ul style="list-style-type: none"> zynq zynqmp versal
<code>generate_keys <key_type></code>	Generate the authentication keys. Options are: <ul style="list-style-type: none"> <code>pem</code> <code>rsa</code> <code>obfuscatedkey</code> 	<ul style="list-style-type: none"> zynq zynqmp versal
<code>h, help</code>	Prints out help summary.	<ul style="list-style-type: none"> All
<code>image <filename(.bif)></code>	Provides a boot image format (.bif) file name.	<ul style="list-style-type: none"> All

Table 42: Bootgen Command and Descriptions (cont'd)

Commands	Description and Options	Used by
<code>log<level_type></code>	Generates a log file at the current working directory with following message types: <ul style="list-style-type: none"> • error • warning (default) • info • debug • trace 	• All
<code>nonbooting</code>	Create an intermediate boot image.	• zynq • zynqmp • versal
<code>o <filename></code>	Specifies the output file. The format of the file is determined by the file name extension. Valid extensions are: <ul style="list-style-type: none"> • .bin (default) • .mcs • .pdi 	• All
<code>overlay_cdo <filename></code>	CDO overlay option provides a way to modify CDO files after they are generated.	versal
<code>p <partname></code>	Specify the part name used in generating the encryption key.	• All
<code>padimageheader <option></code>	Pads the image headers to force alignment of following partitions. Options are: <ul style="list-style-type: none"> • 0 • 1 (default) 	• zynq • zynqmp
<code>process_bitstream <option></code>	Specifies that the bitstream is processed and outputs as .bin or .mcs. <ul style="list-style-type: none"> • For example, if encryption is selected for bitstream in BIF file, the output is an encrypted bitstream. 	• zynq • zynqmp
<code>read <options></code>	Used to read boot headers, image headers, and partition headers based on the options. <ul style="list-style-type: none"> • <code>bh</code>: To read boot header from bootimage in human readable form • <code>iht</code>: To read image header table from bootimage • <code>ih</code>: To read image headers from bootimage. • <code>pht</code>: To read partition headers from bootimage • <code>ac</code>: To read authentication certificates from bootimage 	• zynq • zynqmp • versal

Table 42: Bootgen Command and Descriptions (cont'd)

Commands	Description and Options	Used by
<code>split <options></code>	<p>Splits the boot image into partitions and outputs the files as <code>.bin</code> or <code>.mcs</code>.</p> <ul style="list-style-type: none"> • Bootheader + Image Headers + Partition Headers + <code>Fsbl.elf</code> • <code>Partition1.bit</code> • <code>Partition2.elf</code> 	<ul style="list-style-type: none"> • <code>zynq</code> • <code>zynqmp</code> • <code>versal</code>
<code>spksignature <filename></code>	Generates an SPK signature file.	<ul style="list-style-type: none"> • <code>zynq</code> • <code>zynqmp</code> • <code>versal</code>
<code>verify <filename></code>	This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image is verified against the available partitions.	<ul style="list-style-type: none"> • <code>zynq</code> • <code>zynqmp</code> • <code>versal</code>
<code>verify_kdf</code>	This option is used to validate the Counter Mode KDF used in Bootgen for generation AES keys.	<ul style="list-style-type: none"> • <code>zynqmp</code> • <code>versal</code>
<code>w <option></code>	<p>Specifies whether to overwrite the output files:</p> <ul style="list-style-type: none"> • <code>on</code> (default) • <code>off</code> <p>Note: The <code>-w</code> without an option is interpreted as <code>-w on</code>.</p>	<ul style="list-style-type: none"> • All
<code>zynqmpes1</code>	Generates a boot image for ES1 (1.0). The default padding scheme is ES2 (2.0).	<ul style="list-style-type: none"> • <code>zynqmp</code>

Boot Time Security

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

Secure and Non-Secure Modes in Zynq 7000 SoC Devices

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

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

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

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

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

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

Zynq UltraScale+ MPSoC Device Security

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

See the section "Security" in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) for more information.

Versal Adaptive SoC Security

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

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

Using Encryption

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

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

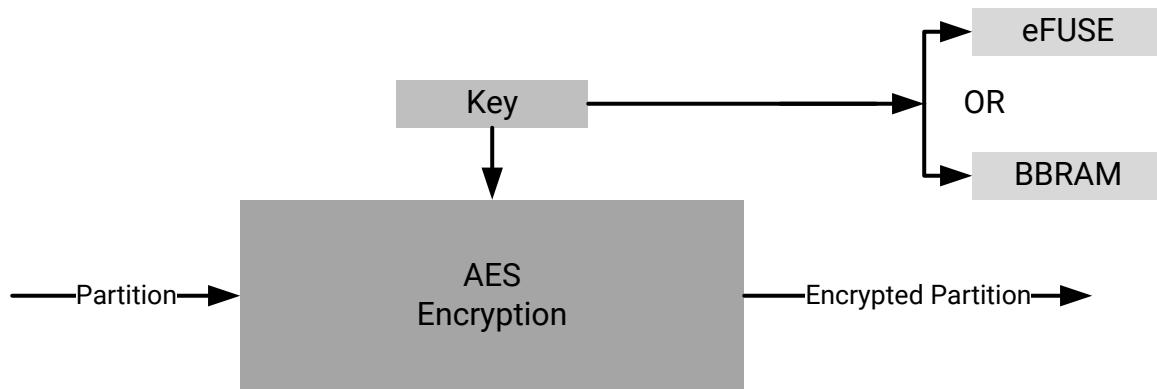
AES is an iterated symmetric block cipher, which means that it does the following:

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

Encryption Process

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

Figure 51: Encryption Process Diagram

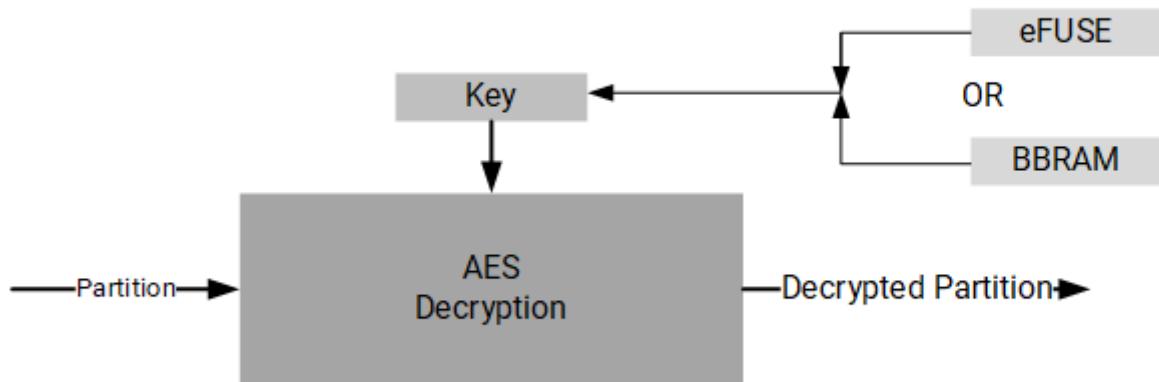


X21274-062320

Decryption Process

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

Figure 52: Decryption Process Diagram



XZ1274-062320

Encrypting Zynq 7000 Device Partitions

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

Example BIF File

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

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

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

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

Key Generation

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

Figure 53: Sample Key File

```
Device      xc7z020clg484;
Key 0       f878b838d8589818e868a828c8488808
Key StartCBC 5C9D95ECBFEC8A1F12A8EB312362C596
Key HMAC    00001112222333444555566667777
```

Encrypting Zynq MPSoC Device Partitions

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

Operational Key

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

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

```
image:
{
    [fsbl_config] opt_key
    [keysrce_encryption] bbram_red_key

    [bootloader,
        destination_cpu = a53-0,
        encryption      = aes,
        aeskeyfile      = aes_p1.nky]fsbl.elf

    [destination_cpu = a53-3,
     encryption      = aes,
     aeskeyfile      = aes_p2.nky]hello.elf
}
```

The operation key is given in the AES key (.nky) file with name Key_Opt as shown in the following example.

Figure 54: Operational Key

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

Bootgen generates the encryption key file. The operational key opt_key is then generated in the .nky file, if opt_key has been enabled in the BIF file, as shown in the previous example.

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

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

Rolling Keys

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

```
image:
{
    [keysrc_encryption] bbram_red_key
    [
        bootloader,
        destination_cpu = a53-0,
        encryption      = aes,
        aeskeyfile      = aes_p1.nky,
        blocks          = 1024(2);2048;4096(2);8192(2);4096;2048;1024
    ] fsbl.elf
    [
        destination_cpu = a53-3,
        encryption      = aes,
        aeskeyfile      = aes_p2.nky,
        blocks          = 4096(1);1024
    ] hello.elf
}
```

Note:

- Number of keys in the key file should always be equal to the number of blocks to be encrypted.
 - If the number of keys are less than the number of blocks to be encrypted, Bootgen returns an error.
 - If the number of keys are more than the number of blocks to be encrypted, Bootgen ignores (does not read) the extra keys.
- If you want to specify multiple Key/IV Pairs, you should specify no. of blocks + 1 pairs
 - The extra Key/IV pair is to encrypt the secure header.
 - No Key/IV pair should be repeated in any of the aes key files given in a single bif except the Key0 and IV0.

Gray/Obfuscated Keys

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

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

Bootgen does the following while creating an image:

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

Another example of using the gray/family key is found in [Use Cases and Examples](#).

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

Key Generation

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

Note:

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

Key Generation

A sample key file is shown below.

Figure 55: Sample Key File

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

Obfuscated Key Generation

Bootgen can generate the obfuscated key by encrypting the red key with the family key and a user-provided IV. The family key is delivered by the AMD Security Group. For more information, see [familykey](#). To generate an obfuscated key, Bootgen takes the following inputs from the BIF file.

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

The command to generate the obfuscated key is:

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

Black/PUF Keys

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

```
image:
{
    [puf_file] pufdata.txt
    [bh_key_iv] black_iv.txt
    [bh_keyfile] black_key.txt
    [fsbl_config] puf4kmode, shutter=0x0100005E, pufhd_bh
    [keysrc_encryption] bh_blk_key

    [
        bootloader,
        destination_cpu = a53-0,
        encryption      = aes,
        aeskeyfile      = aes_p1.nky
    ] fsbl.elf

    [
        destination_cpu = r5-0,
        encryption      = aes,
        aeskeyfile      = aes_p2.nky
    ] hello.elf
}
```

For another example of using the black key, see [Use Cases and Examples](#).

Multiple Encryption Key Files

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

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

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

The following snippet shows a sample encryption key file:

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

    ]fsbla53.elf

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

    ]hello.elf
}
```

- The partition `fsbla53.elf` is encrypted using the keys from `key_p0.nky` file.
- Assuming `hello.elf` has three partitions because it has three loadable sections, then partition `hello.elf.0` is encrypted using keys from the `test2.nky` file.
- Partition `hello.elf.1` is then encrypted using keys from `test2.1.nky`.
- Partition `hello.elf.2` is encrypted using keys from `test2.2.nky`.

Encrypting Versal Device Partitions

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

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

Key Management

Good key management practice includes minimizing the use of secret or private keys. This can be accomplished by using different key/IV pairs across different partitions in the boot image. The result is that the AES key stored on the device, in either the BBRAM or eFUSES, is used for only 384 bits, which significantly limits its exposure to side channel attacks.

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

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

Rolling Keys

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

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

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

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

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

```
image
{
    name = lpd, id = 0x4210002
    partition
    {
        id = 0x0C, type = cdo,
        encryption = aes,
        keysrc = 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
    }
}
```

Note:

- Number of keys in the key file should always be equal to the number of blocks to be encrypted.
- If the number of keys are less than the number of blocks to be encrypted, Bootgen returns an error.
- If the number of keys are more than the number of blocks to be encrypted, Bootgen ignores the extra keys.

Key Generation

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

- If one encryption file is specified and others are generated, Bootgen can make sure to use the same Key0/IV0 pair for the generated keys as in the encryption file for the first partition.
- If an encryption file is generated for the first partition and other encryption file with Key0/IV0 is specified for a later partition, then Bootgen exits and returns the error that an incorrect Key0/IV0 pair was used.
- If no key file is specified and encryption is opted for a partition, Bootgen by default generated an `aes` key file with the name of the partition. By doing this, Bootgen makes sure that a different `aeskeyfile` is used for each partition.
- Bootgen enables the usage of unique key files for each of the partition created due to multiple loadable sections by reading/generating key file names appended with ".1", ".2"..."n" and so on in the same directory of the key file meant for that partition.

Black/PUF Keys

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

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

Meta Header Encryption

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

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

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

The following conditions apply.

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

Using Authentication

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

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

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

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

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

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

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

Bootgen can create a authentication certificate in two ways:

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

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

The following is an example BIF file:

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

For device-specific Authentication information, see the following:

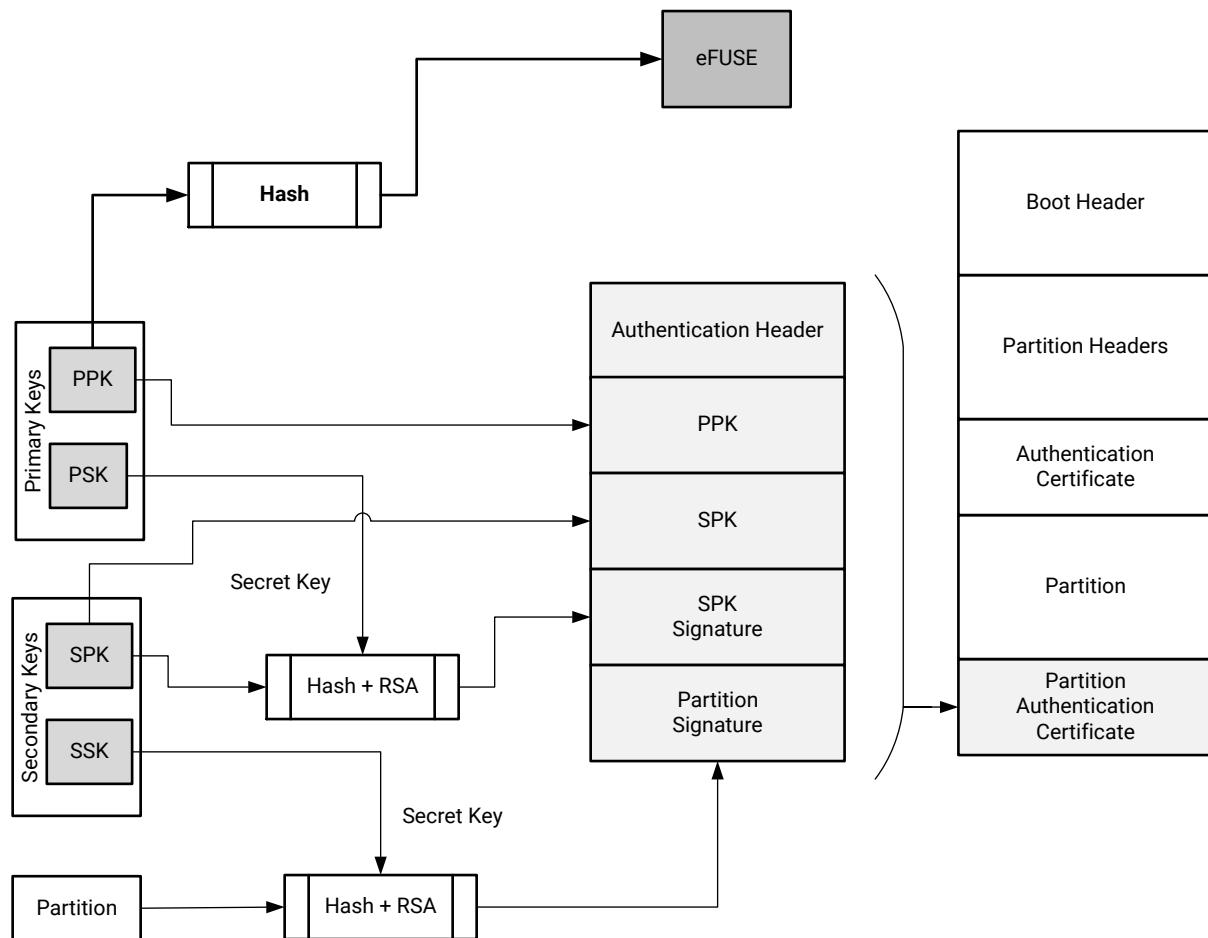
- [Zynq 7000 Authentication Certificates](#)
- [Zynq UltraScale+ MPSoC Authentication Certificates](#)
- [Versal Adaptive SoC Authentication Certificates](#)

Signing

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

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

Figure 56: RSA Partition Signature



The following table shows the options for Authentication.

Table 43: Supported File Formats for Authentication Keys

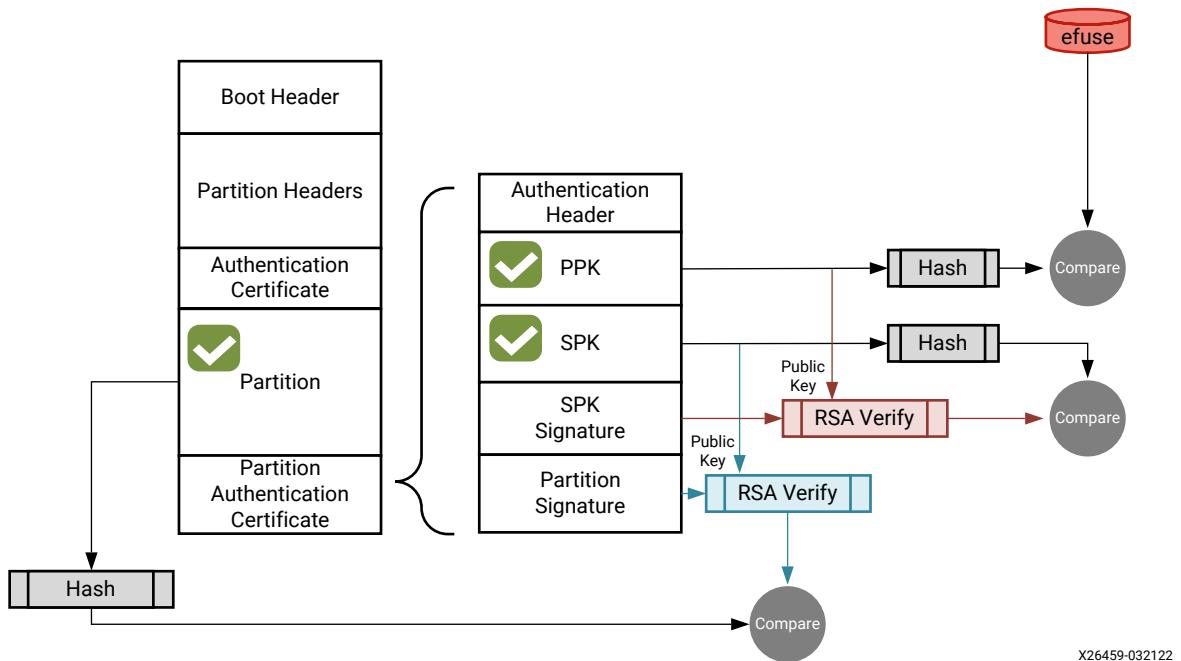
Key	Name	Description	Supported File Format
PPK	Primary Public Key	This key is used to authenticate a partition. It should always be specified when authenticating a partition.	*.pem *.pub
PSK	Primary Secret Key	This key is used to authenticate a partition. It should always be specified when authenticating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

Verifying

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

1. Verify PPK: This step establishes the authenticity of primary key, which is used to authenticate secondary key.
 - a. PPK is read from AC in boot image
 - b. Generate PPK hash
 - c. Hashed PPK is compared with the PPK hash retrieved from eFUSE
 - d. If same, then primary key is trusted, else secure boot fail
2. Verify secondary keys: This step establishes the authenticity of secondary key, which is used to authenticate the partitions.
 - a. SPK is read from AC in boot image
 - b. Generate SPK hashed
 - c. Get the SPK hash, by verifying the SPK signature stored in AC, using PPK
 - d. Compare hashes from step (b) and step (c)
 - e. If same, then secondary key is trusted, else secure boot fail.
3. Verify partitions: This step establishes the authenticity of partition which is being booted.
 - a. Partition is read from the boot image.
 - b. Generate hash of the partition.
 - c. Get the partition hash, by verifying the Partition signature stored in AC, using SPK.
 - d. Compare the hashes from step (b) and step (c)
 - e. If same, then partition is trusted, else secure boot fail

Figure 57: Verification Flow Diagram



X26459-032122

Bootgen can create an authentication certificate in two ways:

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

Zynq UltraScale+ MPSoC Authentication Support

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

NIST SHA-3 Support

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

The generated signature uses the Keccak-SHA3 or NIST-SHA3 based on following table:

Table 44: Authentication Signatures

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

Examples

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

```
image:
{
    [fsbl_config] bh_auth_enable
    [auth_params] ppk_select=0; spk_id=0x00000000
    [pskfile] primary_4096.pem
    [sskfile] secondary_4096.pem
    [pmufw_image] pmufw.elf
    [bootloader, authentication=rsa, destination_cpu=a53-0] fsbl.elf
    [authentication=rsa, destination_cpu=r5-0] hello.elf
}
```

Example 2: BIF file for authenticating the partitions with separate secondary key for each partition:

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

    // FSBL (Partition-0)
    [
        bootloader,
        destination_cpu = a53-0,
        authentication = rsa,
        spk_id = 0x01,
        sskfile = secondary_p1.pem
    ] fsbla53.elf

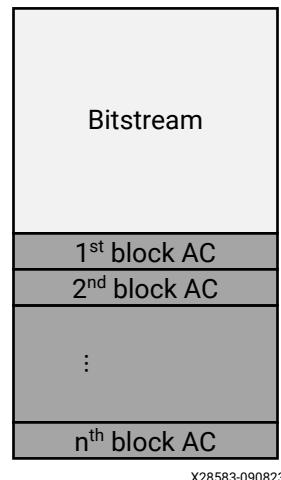
    // ATF (Partition-1)
    [
        destination_cpu = a53-0,
        authentication = rsa,
        exception_level = el-3,
        trustzone = secure,
        spk_id = 0x02,
        sskfile = secondary_p2.pem
    ] bl31.elf

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

Bitstream Authentication Using External Memory

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

Figure 58: Bitstream Authentication Using External Memory



User eFUSE Support with Enhanced RSA Key Revocation

Enhanced RSA Key Revocation Support

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

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

This is achieved by using USER_FUSE0 to USER_FUSE7 eFUSES with the BIF parameter [spk_select](#).

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

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

```
the_ROM_image:
{
    [auth_params]ppk_select = 0
    [pskfile]psk.pem
    [sskfile]ssk1.pem
    [
        bootloader,
        authentication = rsa,
        spk_select = spk-efuse,
        spk_id = 0x8,
        sskfile = ssk2.pem
    ] zynqmp_fsbl.elf
    [
        destination_cpu = a53-0,
        authentication = rsa,
        spk_select = user-efuse,
        spk_id = 0x100,
        sskfile = ssk3.pem
    ]
}
```

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

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

Partitions loaded by CSU ROM always uses `spk_efuse`.

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

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

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

Key Generation

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

The figure shows the sample RSA private key file.

Figure 59: Sample RSA Private Key File

```
-----BEGIN RSA PRIVATE KEY-----
MIJKAIBAAKCAgEA4ppimme6TvPT5+JB2CgXQLU9AyStbnEr21EJu+ZpR9HZ5Plq
6Kb0cFuV6q3EKvISPJyMS0yHpv/11/uTPxyUT6Im5goMyaskz0PS3xTWuYoSDba
YD5021Pi5yBrswWvys6YcIbLTb2k+o86o0Rr/sdQtLR0pbsLfuBFoKMEsK19N12k
E116DM1Tjh9KSpZOzmj7yew2Rm857QqOp8sulVi4qdtIr58+MoQxeETeHcN+zuq4
drlUsUqX3msVb9z0rRwYrBVsksWr5d+xj+cAUpiPjeMGRxg00L6gEGGPTjnqQtG
YFCoCFcBL4JknHF/yMyV7f6wh2xtkKbme+Kuovcz/pQVKEGELkQ9kjweBf5c8Vm
b13NvkrAUOXLYM+py0uY/PGjtz6B5W964LocrT+TRROi4FGotYzk2XmJtODO5dYH
Lw58IOT3zAYwaC/98bUDGYP6kJ9+YqprerLm2U55Ew30PPodjHYihLmBjlpvmu4g
oZ9tXJPch/uRk/tv3e53P2JhWKwdB72FUi8hEgSkCWVAffJwCVFwATettzGlhtz+
Ww3eBAQi6YERwxOOLopaRQzPaC/8XG8u0bTe3MdvsJK/II0AqVnT17Dfs
QKzT2ap8+Iwx/vuaWAIld0qYCDKKM1GGz5bQhEgRnk0/1/pOK1lPRL8wH0CAwEA
AQKCAgA3qhscuOxgZq8gYEky67G4pgUks0PSK7n3qXqNM17FvtToO/oPJHUYgz
PPpaXmRHCgNsH+GWChM08gDU8pKwR0jPZolyTpkfVDIC/M6KI+luEZ9E
iZkbQgNb+4Ig6kvYzO2/gR2za6Rn0shli3q4F4mYkVYX5NQXmI/Doa2ph1AnDQX
roI0bnvvYoSvppHynXIKU7UTMutPR1sdhpuFYMxjnP0imrAzofU3FA7Y
eU+ryghk2ekJpL3TKTzqZ3mh85A8FOyrQfPtWZ1/6A0nInF6apclxHgGQKm2WoEV
DZ/vekYcq00GK1+qtkDvqx5tEaX1KG1c0PBWg5acfkpNZ0K0wOG6iCueNvATcJ9
RoMq7c7zZOYh4SzWgSjP3a8neGcnhG0T6BGYCGjPXWR2Y6ri/71rDcOBVSc3zS8p
IVKABp13PIg2lhMnxdc60RPh8dhOUtua3+1SyGx37Ad926OUeHHJIpz28DkzTg
CY7RUSSDSh6wDuDbheli4nzZDGhKq9eAzXGzhIn0zcxpWvG54uHTHNNqBEFJ2S
ZSJ8sq4aYiZCiW/FrqKgg8wBygKcetr3/LcAm4r3p19mHk1555QQNdpk+ba+3GLp
bEy0869KwCyPKfWY5p16VNglYcxe/TofMDCHQARA2wLrnN1sQQKCAQEA80nn83su
OYN90c22owfm/MHGJ6mPi5LpRtGy1WbcAbDsZs7rjQ4Iz46JQlMpiQ10IpNbVub7
sW0FUK7sVo0X2MSl+Eps2Dq021+7hY6+MGALtPpg9n2Jz91fCyVXfNqv5SiMv6Te
6/jur69KiwhztYf17JR4GGUdcCWyAMTdg3pQDDH99Vp436k1vk41MyjeQaIp0/
Fzkik1fyN84j9jvtagoMk0fzzckioGGSs4ci0ds3DEgGC9x1hDkIs9UFPk1Pfw+7
qYnsT7XIwoTCBrvQ11Kp5fL2UhSRsIQV82u44IPfcU3xWgeyInSGx0RFsv5Rwov
v9sJFVsF1X5EQQKCAQEA7nFNK5gbPKA0nxKTeM1ZMhp99/YqRxpj5irmXmrF54cn
sZPpG/dvbJBXILAd9heSYjw8FNY5ehJhL9IqzEVavFr8SAvu2FyI9MN0d9wUvpJG
55JxX9K090uSzaXZVimV/5xumbnynwx2Zwgxs1SAYoNy+8scv1z1XQxZzeUaohaM
VVuL1HdRzE0afrfFsnfugID172Mb14t2cKTFTek/iYAvF9bk076upkPmMu4V7yFT
of9QFkq8qBrthEpvaKNTObpU5TrzskxUH3rYXVnAZgpEXEJdeVVFYzSLf4SC45mx8
GFP3pYtPBKVrUesuEvQ70IeoicGRXFgC9TPmYwrQKCAQEA5D1CoPbAD+7ejVsD
a4FFx2K+7rin86A1q9h1zAK0n6jhyzeRpq1h7jgFiaj9hRnppVx3pdSD+6DJch
UTV+a+fcuMnBVGQk/3+ZYhvfK2z/rqJyUuzFXdWYR0ANz7GY5seKDC2fhGEg0dV
DIg6XV5sGvsuQJyj+HE0xoSdP1Cxe9fyNrWEgvQkzgx64gXlmvXZPbs//F3EIne
6801kyz3d1LEJ2wJ3V2pdc0BnvE4175K1/f9zCTgDtKe6m7/Q0qu0MreyJf/HWyy
UmLP0BdlAogfdIkApR0rKvym7mlGQUMWXaq8sTS1FpPxWYI4TpFwi2aXAg2a9w3
qdKVs0QKCAQH8nolcFT/mxulsBY9ikDSRvPBoU6qe8UPC3zNmowy25nv8jD/opp
iLgxjdLMkuieJ7ajluwq8GbQ5iL2cEftrs8yR9L/SG0HcEs0QjKDZzAuHDoIVNuAS
CoS2dse4nv26zjn1Os2BvmHvvu13/BvtJFrKrUeS8MT/KZ3jabD6nbEkhGX+m25c
JhvLhnA6pMoBlM1MzWu8vH/FVCoEqxwUfRjzhy6BlRuoHwIacOg9CvffltcImy
cc+F7mvl/rB3X6GWJ52N+9S/UDXfsXF2wA9q171gYE5DL/fD1+bb7GI+fK8VCHZ
2P01bCtIMF5oxVu28fdx9r7TcxhdL2VAoIBADmGyfxvgEqhALqdWZQmtRRNisWQ
y0/RfED7dNtN8o5vjBCbrOV/tQ3Ddbb7a0kw01NFrixR7Kiki98SkKN0EiCprfc
+ccs6kAST2cPH/nGG91br0Am9FOG2q5cX6kDK1lhqHe+1UYm/34a+2wN0/CwAh7MH
gECABtqx9QCD/DJ1+n5ocrYk5RsQJrtnwoP4L8X24dRiMiRMIss4V9uuyRLQTwV/
k3TOjRgL5eRKbcVwV7c8kmaGDWfM/eVLLQW+wEaUwY+TdSUhlyvgsG5yijkhCAEe
/+Az0w5Zu1vnLbj5eXKu1LWIS1OsDCBfJepuINHoUpBwsGzFb7ZxtpK2X1M=
-----END RSA PRIVATE KEY-----
```

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

BIF Example

A sample BIF file, generate_pem.bif:

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

Command

The command to generate keys is, as follows:

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

PPK Hash for eFUSE

Bootgen generates the PPK hash for storing in eFUSE for PPK to be trusted. This step is required only for Asymmetric HW Root-of-Trust (AHWRoT), and can be skipped for AHWRoT for the AMD Zynq™ UltraScale+™ MPSoC device. The value from `efuseppksha.txt` can be programmed to eFUSE for AHWRoT.

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

BIF File Example

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

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

Command

The command to generate PPK hash for eFUSE programming is:

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

Versal Authentication Support

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

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

Meta Header Authentication

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

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

Authentication Optimization

Starting in v2024.1, For a Versal adaptive SoC, you can opt for authentication time reduction.

If you are using the same SPK (with the same authentication algorithm and key strength) for the Meta Header and partitions loaded by the PLM, it is possible to avoid the SPK and partition data signature validations. You can enable the use of authenticated partition digests that are stored in the optional Image Header Table. The partition digests are SHA digests that are used to validate the integrity of the partition data at boot.

When reading the Meta Header, PLM authenticates and copies the table of partition digests to be used during partition loading. Authentication can be skipped for any partition with a digest entry in the table. In this case, the partition data is passed through the SHA engine, and the resulting digest is compared against the authenticated digest value stored for that partition in the Image Header Table.

For enabling authentication optimization refer to [enable_auth_opt](#).

PPK Hash for eFUSE

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

BIF File Example

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

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

Command

The command to generate PPK hash for eFUSE programming is:

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

Cumulative Secure Boot Operations for Versal Adaptive SoC

Table 45: Cumulative Secure Boot Operations

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

Versal Hashing Scheme

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

The hashing scheme as per the following table:

Table 46: Partition Chunking Scheme

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

The SECURE_CHUNK_SIZE applicable to AMD Versal™ is 32 KB.

Note: For encryption use cases, do that the user key rolling is wholly contained within a hash chunk.

Using HSM Mode

In current cryptography, all the algorithms are public, so it becomes critical to protect the private/secret key. The hardware security module (HSM) is a dedicated crypto-processing device that is specifically designed for the protection of the crypto key lifecycle. This module increases key handling security, because only public keys are passed to the Bootgen and not the private/secure keys.

In some organizations, an infosec staff is responsible for the production release of a secure embedded product. The infosec staff might use the HSM for digital signatures and a separate secure server for encryption. The HSM and secure server typically reside in a secure area. The HSM is a secure key/signature generation device that generates private keys, signs the partitions using the private key, and provides the public part of the RSA key to Bootgen. The private keys reside in the HSM only.

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

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

For more information about the HSM mode for FPGAs, see the [HSM Mode](#).

Using Advanced Key Management Options

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

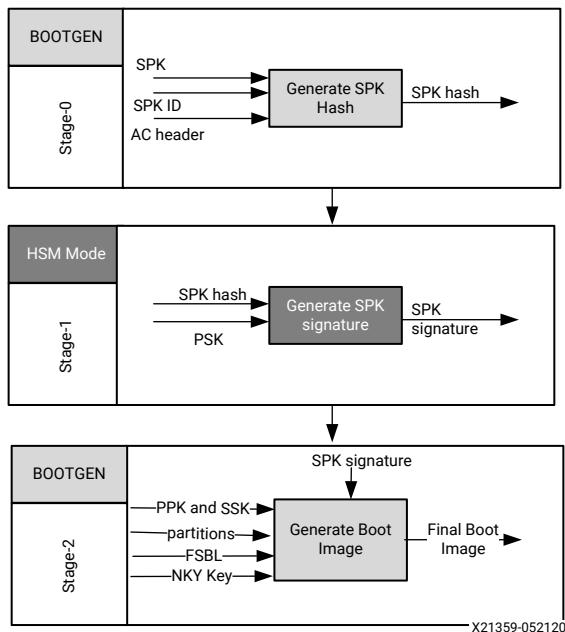
Note: HSM flow is not supported for mcs format boot image generation.

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

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

This figure uses the AMD Zynq™ UltraScale+™ MPSoC to illustrate the stages.

Figure 60: Generic 3-Stage Boot Image



Boot Process

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

```
all:
{
  [auth_params] ppk_select=1;spk_id=0x8
  [keysrc_encryption]bbram_red_key
  [pskfile]primary.pem
  [sskfile]secondary.pem
  [
    bootloader,
    encryption=aes,
    aeskeyfile=aes.nky,
    authentication=rsa
  ]fsbl.elf
  [destination_cpu=a53_0,authentication=rsa]hello_a53_0_64.elf
}
```

Stage 0: Create a Boot Image using HSM Mode

A trusted individual creates the SPK signature using the Primary Secret Key. The SPK signature is on the Authentication Certificate Header, SPK, and SPK ID. To generate a hash for the above, use the following BIF file snippet.

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

The following is the Bootgen command:

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

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

Stage 1: Distribute the SPK Signature

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

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

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

Stage 2: Encrypt using AES in FSBL

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

- The SPK Signature from the trusted individual.
- The SSK, SPK, and SPKID

```
Stage2:
{
    [keysrc_encryption]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
}
```

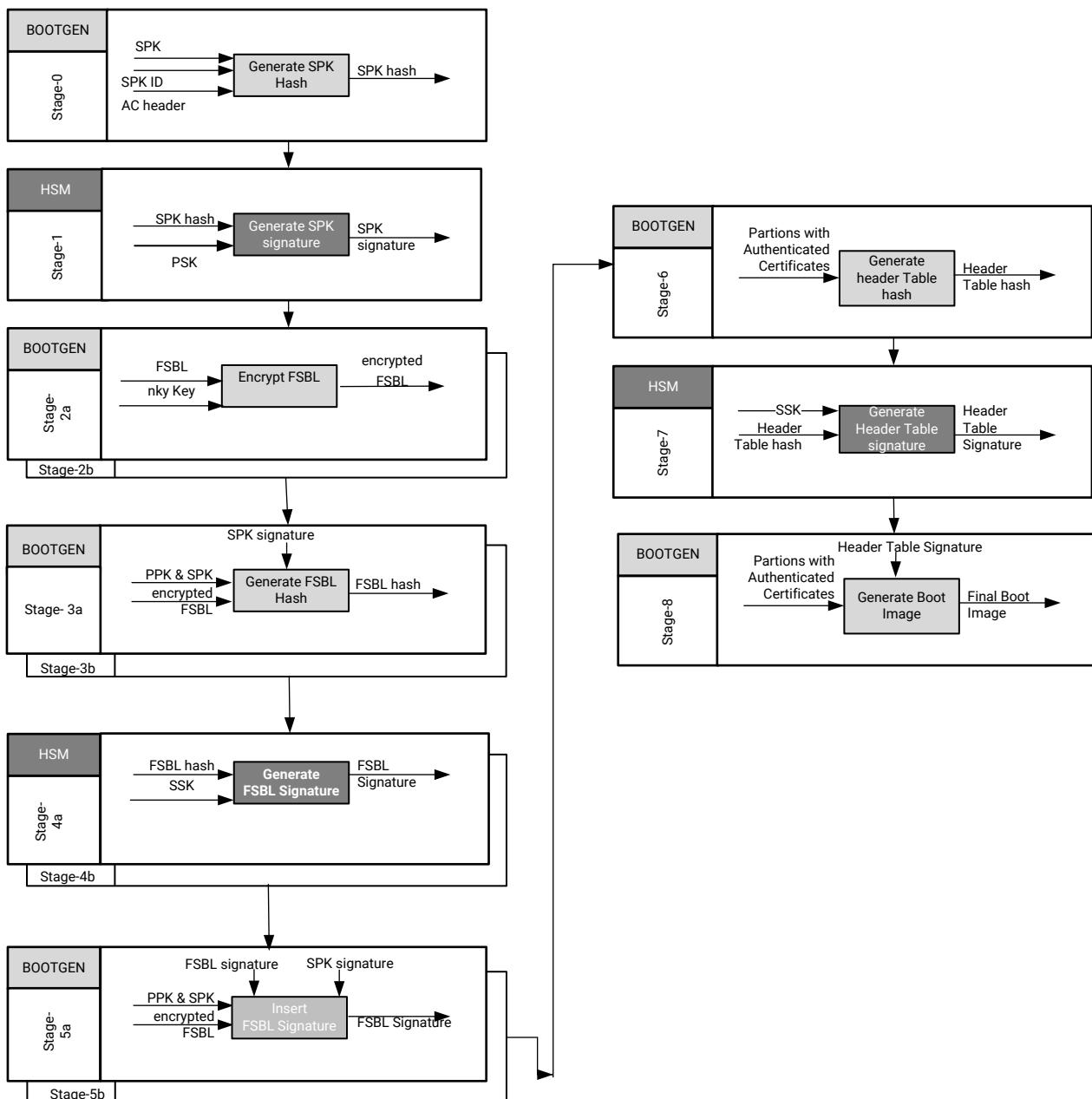
The Bootgen command is:

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

Creating a Zynq 7000 SoC Device Boot Image using HSM Mode

The following figure provides a diagram of an HSM mode boot image for an AMD Zynq™ 7000 SoC device. The steps to create this boot image are immediately after the diagram.

Figure 61: Stage 0 to 8 Boot Process



X21416-052120

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

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

Stage 0: Generate a Hash for SPK

This stage generates the hash of the SPK key.

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

The following is the Bootgen command.

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

Stage 1: Sign the SPK Hash

This stage creates the signatures by signing the SPK hash

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

Or by using the following OpenSSL program.

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

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

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

Stage 2: Encrypt using AES

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

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

The Bootgen command is as follows.

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

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

Stage 3: Generate Partition Hashes

This stage generates the hashes of different partitions.

Stage 3a: Generate the FSBL Hash

The BIF file is as follows:

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

The Bootgen command is as follows.

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

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

Stage 3b: Generate the bitstream hash

The stage3b BIF file is as follows:

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

The Bootgen command is as follows.

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

The output is the hash file `system.bit.0.sha256`.

Stage 4: Sign the Hashes

This stage creates signatures from the partition hash files created.

Stage 4a: Sign the FSBL partition hash

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

Or by using the following OpenSSL program.

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

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

Stage 4b: Sign the bitstream hash

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

Or by using the following OpenSSL program.

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

The output is the signature file `system.bit.0.sha256.sig`.

Stage 5: Insert Partition Signatures

Insert partition signatures created above are changed into authentication certificates.

Stage 5a: Insert the FSBL signature

The `stage5a.bif` is as follows.

```
stage5a:  
{  
    [ppkfile] primary.pub  
    [spkfile] secondary.pub  
    [spksignature] secondary.pub.sha256.sig  
    [bootimage, authentication=rsa, presign=zynq-fsbl-0.elf.0.sha256.sig]  
    fsbl_e.bin  
}
```

The Bootgen command is as follows.

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

The authenticated output files are `fsbl_e_ac.bin` and `efuseppkbts.txt`.

Stage 5b: Insert the bitstream signature

The `stage5b.bif` is as follows.

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

The Bootgen command is as follows.

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

The authenticated output file is `system_e_ac.bin`.

Stage 6: Generate Header Table Hash

This stage generates the hash for the header tables.

The `stage6.bif` is as follows.

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

The Bootgen command is as follows.

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

The output hash file is `ImageHeaderTable.sha256`.

Stage 7: Generate Header Table Signature

This stage generates the header table signature.

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

Or by using the following OpenSSL program:

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

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

Stage 8: Combine Partitions, Insert Header Table Signature

The `stage8.bif` is as follows:

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

The Bootgen command is as follows:

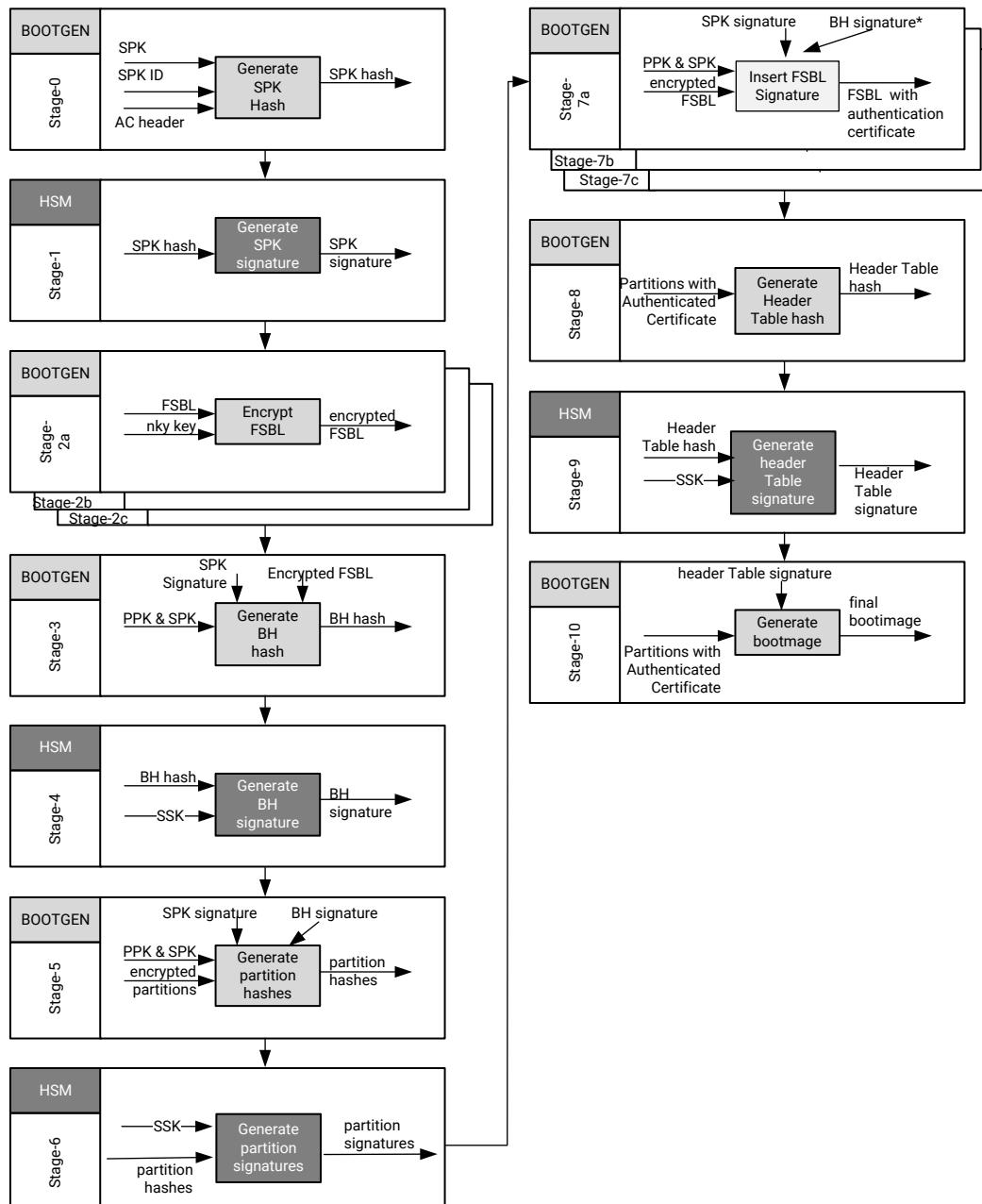
```
bootgen -image stage8.bif -w -o final.bin
```

The output is the boot image file `final.bin`.

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

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

Figure 62: 0 to 10 Stage Boot Process



X21547-052120

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

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

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

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

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

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

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

Stage 0: Generate a hash for SPK

The following is the snippet from the BIF file.

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

The following is the Bootgen command:

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

Stage 1: Sign the SPK Hash (encrypt the partitions)

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

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

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

Stage 2a: Encrypt the FSBL

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

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

The Bootgen command is:

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

Stage 2b: Encrypt Bitstream

Generate the following BIF file entry:

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

The Bootgen command is:

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

Stage 3: Generate Boot Header Hash

Generate the boot header hash using the following BIF file:

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

The Bootgen command is:

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

Stage 4: Sign Boot Header Hash

Generate the boot header hash with the following OpenSSL command:

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

Stage 5: Get Partition Hashes

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

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

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

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

The Bootgen command is:

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

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

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

Stage 6: Sign Partition Hashes

Create the following files using OpenSSL:

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

Stage 7: Insert Partition Signatures into Authentication Certificate

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

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

The Bootgen command is as follows:

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

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

```
stage7b:
{
    [ppkfile]primary.pub
    [spkfile]secondary.pub
    [spksignature]secondary.pub.sha384.sig
```

```
[bhsignature]boothdr.sha384.sig
[
  bootimage,
  authentication=rsa,
  presign=system.bit.0.sha384.sig
] system_e.bin
}
```

The Bootgen command is:

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

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

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

The Bootgen command is:

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

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

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

The Bootgen command is:

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

Stage 8: Combine Partitions, Get Header Table Hash

Enter the following in a BIF file:

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

The Bootgen command is:

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

Stage 9: Sign Header Table Hash

Generate the following files using OpenSSL:

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

Stage 10: Combine Partitions, Insert Header Table Signature

Enter the following in a BIF file:

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

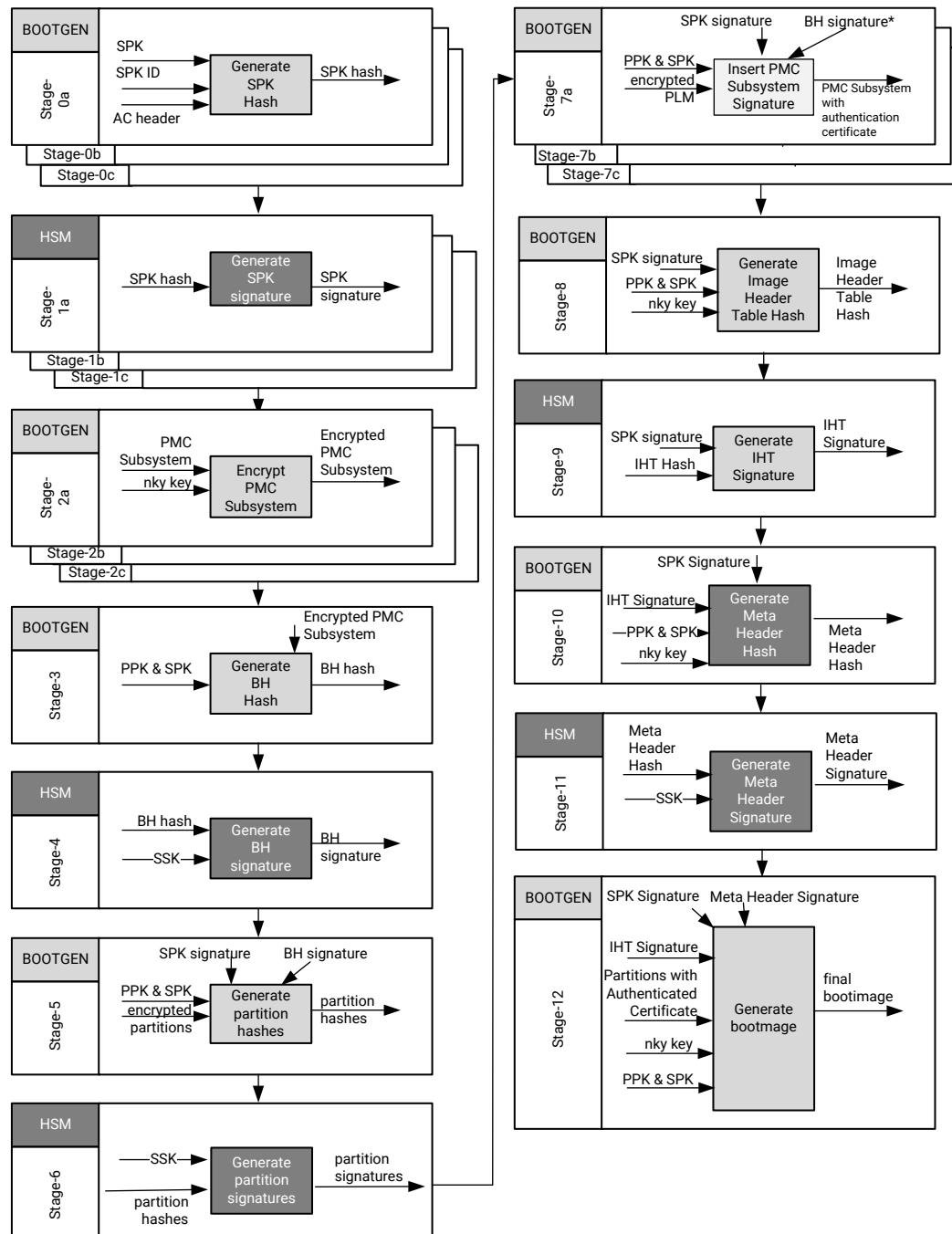
The Bootgen command is:

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

Creating a Versal Device Boot Image Using HSM

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

Figure 63: 0 to 12 Stage Boot Process



X21547-111020

Note: The PMC subsystem includes PLM, PMC_CDO.

Generating the PDI

Generate the PDI using the standard BIF.

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

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

    metaheader
    {
        authentication = rsa,
        pskfile = rsa-keys/PSK2.pem,
        sskfile = rsa-keys/SSK2.pem
        encryption = aes,
        keysrc = 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 = 0x420c003
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 Mode Steps

Stage 0: Generate SPK Hash

Generate hash for SSK1:

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

stage0_SSK1:
{
    spkfile = rsa-keys/SSK1.pub
}
```

Generate hash for SSK2:

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

stage0_SSK2:
{
    spkfile = rsa-keys/SSK2.pub
}
```

Generate hash for SSK3:

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

stage0_SSK3:
{
    spkfile = rsa-keys/SSK3.pub
}
```

Stage 1: Sign SPK hash

Sign the generated hashes:

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

Stage 2: Encrypt Individual Partitions

Encrypt partition 1:

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

stage2a:
{
    image
    {
```

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

Encrypt partition 2:

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

stage2b_1:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0C, type = cdo,
            encryption=aes, delay_auth,
            keysrc = bbram_red_key,
            aeskeyfile = encr_keys/key1.nky,
            dpacm_enable,
            file = images/gen_files/lpd_data.cdo
        }
    }
}
```

Encrypt partition 3:

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

stage2b_2:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            id = 0x0B, core = psm,
            encryption = aes, delay_auth,
            keysrc = bbram_red_key,
            aeskeyfile = encr_keys/key2.nky,
        }
    }
}
```

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

Encrypt partition 4:

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

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

Encrypt partition 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
        }
    }
}
```

Stage 3: Generate Boot Header Hash

```
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  
    }  
}
```

Stage 4: Sign Boot Header Hash

Sign the generated hashes:

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

Stage 5: Generate Partition Hashes

```
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
    }
}
```

Stage 6: Sign Partition Hashes

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

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

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

Stage 7: Insert Partition Signatures into Authentication Certificates

Insert partition 1 signature:

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

stage7a:
{
    bhsignature = 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
        }
    }
}
```

Insert partition 2 signature:

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

stage7b_1:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
        {
            type = bootimage,
            authentication = rsa,
            ppkfile = rsa-keys/PSK3.pub,
            spkfile = rsa-keys/SSK3.pub,
            spksignature = SSK3.pub.sha384.sig,
            presign = lpd.0.sha384.sig,
            file = lpd_lpd_data_e.bin
        }
    }
}
```

Insert partition 3 signature:

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

stage7b_2:
{
    image
    {
        name = lpd, id = 0x4210002
        partition
```

```
{  
    type = bootimage,  
    authentication = rsa,  
    ppkfile = rsa-keys/PSK1.pub,  
    spkfile = rsa-keys/SSK1.pub,  
    spksignature = SSK1.pub.sha384.sig,  
    presign = psm.0.sha384.sig,  
    file = lpd_psm_fw_e.bin  
}  
}  
}
```

Insert partition 4 signature:

```
command : bootgen -arch versal -image stage7c.bif -o fpd_e_ac.bin -w on -  
log error  
  
stage7c:  
{  
    image  
    {  
        id = 0x1c000000, name = fpd  
        { type = bootimage,  
            authentication=rsa,  
            ppkfile = rsa-keys/PSK3.pub,  
            spkfile = rsa-keys/SSK3.pub,  
            spksignature = SSK3.pub.sha384.sig,  
            presign = fpd_data.cdo.0.sha384.sig,  
            file = fpd_e.bin  
        }  
    }  
}
```

Insert partition 5 signature:

```
command : bootgen -arch versal -image stage7d.bif -o subsystem_e_ac.bin -w  
on -log error  
  
stage7d:  
{  
    image  
    {  
        id = 0x1c000033, name = ss  
        { type = bootimage,  
            authentication = rsa,  
            ppkfile = rsa-keys/PSK2.pub,  
            spkfile = rsa-keys/SSK2.pub,  
            spksignature = SSK2.pub.sha384.sig,  
            presign = ss.0.sha384.sig,  
            file = subsystem_e.bin  
        }  
    }  
}
```

Stage 8: Generate Image Header Table Hash

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

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}
    }
}
```

Stage 9: Sign Image Header Table Hash

Sign the generated hashes:

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

Stage 10: Generate Meta Header Hash

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

stage8b:
{
```

```
headersignature = imageheadertable.sha384.sig
id_code = 0x04ca8093
extended_id_code = 0x01
id = 0x2

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

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

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

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

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

Stage 11: Sign Meta Header Hash

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

Stage 12: Combine Partitions and Insert Header Signature

Build the complete PDI:

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

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

    metaheader
    {
```

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

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

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

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

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

Note: If signing using ecdsa, below is an example for ecdsa-p384 using openssl.

Assuming secondary.pub.sha384 is a Bootgen generated hash for a given SPK, below is a script to generate Bootgen usable signature with a PSK, primary.pem.

```
#!/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 Support

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

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

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

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

Note: PLM elf in each SLR should be same.

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

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

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

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

Example- HBM Devices

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

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

```
        }

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

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

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

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

```
}

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

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

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

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

```

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

```

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

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

```

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

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

    boot_config {smap_width=0,bh_auth_enable}
    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 Support

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

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

Encryption and Authentication

AMD 7 series FPGAs use the embedded, PL-based, hash-based message authentication code (HMAC) and an advanced encryption standard (AES) module with a cipher block chaining (CBC) mode. For UltraScale devices and beyond, AES-256/Galois Counter Mode (GCM) are used, and HMAC is not required.

Encryption Example

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

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

The BIF file looks like the following:

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

Authentication Example

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

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

The BIF file is as follows:

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

Family or Obfuscated Key

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

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

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

Figure 64: AES Key Sample

```
Device xckull5;
EncryptKeySelect BBRAM;
KeyObfuscate 94da9014cb2203f502f81d14fa2471f4a8902b16d9d408c9c66db214c1640db7, 0;
StartIvObfuscate c485144e397a92081ad20c867a005272, 0;
Key0 dcd2e72ad1b281ecca5e0790b65b94090ec1c8fc010eb01e56717345df4c7010, 0;
StartIv0 3fe826e5495dblbdaf0c2ca2e8640911, 0;
KeyObfuscate 967a6d1ecccfd1990241007de18f41d69ca7231852c0061fb6c78e204c5f3, 1;
StartIvObfuscate 7ab9a7ca88474d7f95ed1b548523451b, 1;
Key0 af84947a9cc256c090d5aelc53ed3fd33bb553d7039e445829ba4cffbe56ffe3, 1;
StartIv0 a50026e212363eld71fa6f4fb540ce42, 1;
```

HSM Mode

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

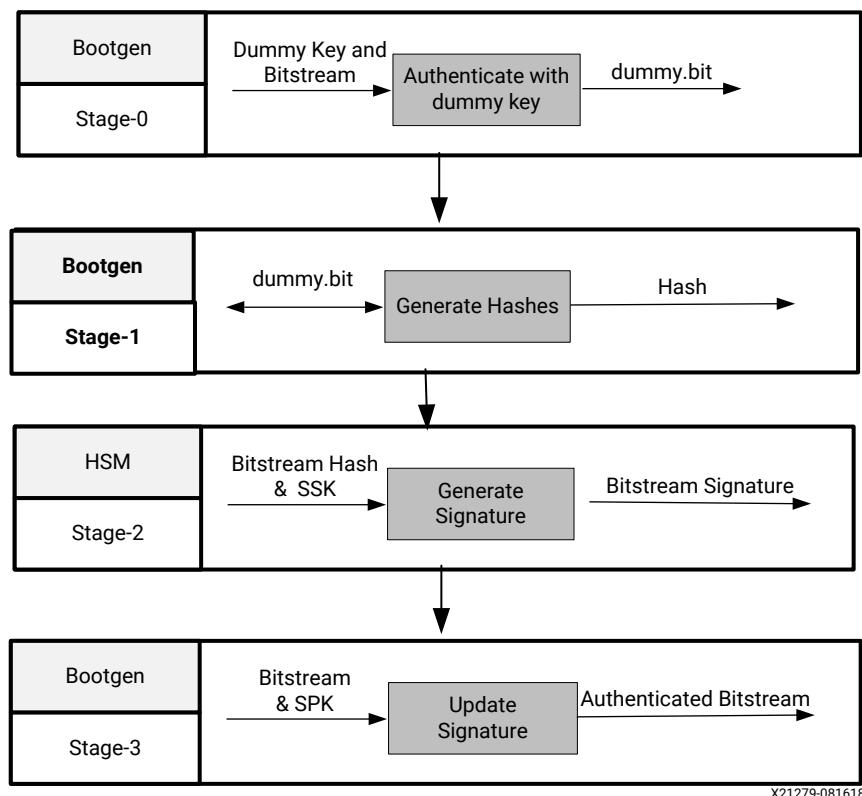
Standard Mode

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

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

The following steps listed below describe how to generate an authenticated bitstream in HSM mode, where the secret keys are maintained by secure team and not available with the user. The following figure shows the HSM mode flow:

Figure 65: HSM Mode Flow



Stage 0: Authenticate with dummy key

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

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

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

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

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

Stage 1: Generate hashes

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

`Stage1.bif` is as follows:

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

Stage 2: Sign the hash HSM

Here, OpenSSL is used for demonstration.

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

Stage 3: Update the RSA certificate with Actual Signature

The `Stage3.bif` is as follows:

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

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

Note: The public key digest, which must be burnt into eFUSES, can be found in the generated `rsaPubKeyInfo.pem.nky` file in Stage3 of HSM mode.

HSM Flow with Both Authentication and Encryption

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

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

```
the_ROM_image:  
{  
[aeskeyfile] encrypt.nky  
[sskfile] dummykey.pem  
[encryption=aes, authentication=rsa, keylife =32] plain-system.bit  
}  
  
bootgen -arch fpga -image stage0.bif -w -o auth-encrypt-system.bit -log info
```

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

Stage 1: Generate hashes

See the following code for an example.

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

Stage 2: Sign the hash HSM

Here, OpenSSL is used for demonstration.

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

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

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

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

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

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

Stage 3: Update the RSA certificate with the actual signature

See the following code for an example.

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

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

Use Cases and Examples

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

Zynq MPSoC Use Cases

Simple Application Boot on Different Cores

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

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

PMU Firmware Load by BootROM

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

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

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

PMU Firmware Load by FSBL

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

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

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

Booting Linux

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

- The `fsbl_a53.elf` is the bootloader and runs on a53-0.
- The `pmu_fw.elf` is loaded by FSBL.
- The `b131.elf` is the Arm® Trusted Firmware (ATF), which runs at el-3.
- The U-Boot program, `uboot`, runs at el-2 on a53-0.
- The Linux image, `image.ub`, is placed at offset `0x1E40000` and loaded at `0x10000000`.

```
the_ROM_image:  
{  
    [bootloader, destination_cpu = a53-0] fsbl_a53.elf  
    [destination_cpu=pmu] pmu_fw.elf  
    [destination_cpu=a53-0, exception_level=el-3, trustzone] b131.elf  
    [destination_cpu=a53-0, exception_level=el-2] u-boot.elf  
    [offset=0x1E40000, load=0X10000000, destination_cpu=a53-0] image.ub  
}
```

Encryption Flow: BBRAM Red Key

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

```
the_ROM_image:  
{  
    [keysrc_encryption] bbram_red_key  
    [  
        bootloader,  
        encryption=aes,  
        aeskeyfile=aes0.nky,  
    ]  
}
```

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

Encryption Flow: Red Key Stored in eFUSE

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

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

Encryption Flow: Black Key Stored in eFUSE

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

```
the_ROM_image:
{
    [fsbl_config] puf4kmode, shutter=0x0100005E
    [auth_params] ppk_select=0; spk_id=0x5
    [pskfile] primary_4096.pem
    [sskfile] secondary_4096.pem
    [keysrc_encryption] efuse_blk_key
    [bh_key_iv] bhkeyiv.txt
    [
        bootloader,
        encryption=aes,
        aeskeyfile=aes0.nky,
        authentication=rsa
    ] fsbl.elf
}
```

Note: Boot image authentication is compulsory for using black key encryption.

Encryption Flow: Black Key Stored in Boot Header

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

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

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

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

Encryption Flow: Gray Key Stored in eFUSE

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

```
the_ROM_image:
{
    [keysrc_encryption] efuse_gry_key
    [bh_key_iv] bh_key_iv.txt

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

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

Encryption Flow: Gray Key Stored in Boot Header

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

```
the_ROM_image:
{
    [keysrc_encryption] bh_gry_key
    [bh_keyfile] bhkey.txt
    [bh_key_iv] bh_key_iv.txt

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

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

Operational Key

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

```
the_ROM_image:
{
    [fsbl_config] opt_key
    [keysrc_encryption] efuse_red_key

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

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

Using Op Key to Protect the Device Key in a Development Environment

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

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

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

The following procedures describe the steps to build an image:

Procedure-1

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

1. Encrypt Bootloader with device key:

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

Example `stage1.bif`:

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

Example `aes.nky` for `stage1`:

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

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

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

Example of `stage2.bif`:

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

    [
        destination_cpu=a53-1,
        encryption=aes,
        aeskeyfile=aes-opt1.nky
    ] hello1.elf
}
```

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

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

Procedure-2

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

1. Encrypt Bootloader with device key:

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

Example `stage1.bif`:

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

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

Example `aes.nky` for `stage1`:

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

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

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

Example of stage2a.bif:

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

Example of stage2b.bif:

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

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

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

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

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

Example of stage3.bif:

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

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

Single Partition Image

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

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

U-Boot Command for Loading Secure Images

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

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

Only Authentication Use Case

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

1. Create a single partition image that is authenticated at U-Boot.

Note: If you provide an `elf` file, it should not contain multiple loadable sections. If your `elf` file contains multiple loadable sections, you should convert the input to the `.bin` format and provide the `.bin` as input in `bif`. An example `bif` is as follows:

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

2. When the image is generated, download the authenticated image to the DDR.
3. Execute the U-Boot command to authenticate the secure image as shown in the following example.

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

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

Only Encryption Use Case

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

Authentication Flow

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

```
the_ROM_image:
{
    [fsbl_config] bh_auth_enable
    [auth_params] ppk_select=0; spk_id=0x00000000
    [pskfile] PSK.pem
    [sskfile] SSK.pem

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

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

BIF File with SHA-3 eFUSE RSA Authentication and PPK0

This example shows how to create a boot image with authentication enabled for FSBL and the application with efuse authentication. This is the default selection. In this process the PPK hash in the boot image is verified with the hash from efuse.

```
the_ROM_image:
{
    [auth_params] ppk_select=0; spk_id=0x00000000
    [pskfile] PSK.pem
    [sskfile] SSK.pem

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

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

XIP

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

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

See [xip_mode](#) for more information about the command.

Split with "Offset" Attribute

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

```
the_ROM_image:
{
    [split]mode=slaveboot,fmt=bin
    [bootloader, destination_cpu = a53-0] fsbl.elf
    [destination_cpu = pmu, offset=0x3000000] pmufw.elf
    [destination_device = pl, offset=0x4000000] design_1_wrapper.bit
    [destination_cpu = a53-0, exception_level = el-3, trustzone,
    offset=0x6000000]\ hello.elf
}
```

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

Versal Adaptive SoC Use Cases

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

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

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

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

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

Bootloader, PMC_CDO

This example shows how to use Bootloader with PMC_CDO.

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

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

Bootloader, PMC_CDO with Load Address

This example shows how to use Bootloader with PMC_CDO and load address.

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

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

Enable Checksum for Bootloader

This example shows how to enable checksum while using bootloader.

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

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

Bootloader, PMC_CDO, PL CDO, NPI

This example shows how to use bootloader with PMC_CDO and NPI.

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

Bootloader, PMC_CDO, PL CDO, NPI, PS CDO, and PS ELFs

This example shows how to use bootloader with PMC_CDO, NPI, PS CDO, and PS ELFs.

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

AI Engine Configuration and AI Engine Partitions

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

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

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

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

Appending New Partitions to Existing PDI

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

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

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

RSA Authentication Example

This example demonstrates the use of RSA authentication.

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

```
        authentication=rsa, 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 Authentication Example

This example demonstrates the use of ECDSA authentication.

```
all:
{
    id_code = 0x04CA8093
    extended_id_code = 0x01
    boot_config {bh_auth_enable}
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {type = bootloader,
            authentication = ecdsa-p384, 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 Encryption Example

This example demonstrates the use of AES Encryption.

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

    image
    {
```

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

AES Encryption with Key Rolling Example

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

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

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

AES Encryption with Multiple Key Sources Example

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

```
all:
{
    bh_keyfile = ./PUF4K_KEY.txt
    puf_file = ./PUFHD_4K.txt
    bh_kek_iv = ./blk_iv.txt
    bbram_kek_iv = ./bbbram_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=bbbram_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=bbbram_blk_key, dpacm_enable, revoke_id = 0x6,
        aeskeyfile = metaheader.nky
    }
    image
```

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

AES Encryption and Authentication Example

This example demonstrates the use of AES encryption and authentication.

```
all:  
{  
    bh_kek_iv = ./blkiv.txt  
    bh_keyfile = ./blkkey.txt  
    efuse_kek_iv = ./efuse_blkIv.txt  
    boot_config {bh_auth_enable, puf4kmode, shutter=0x8100005E}  
    id_code = 0x04CA8093  
    extended_id_code = 0x01  
    image  
{  
        name = pmc_subsys, id = 0x1c000001  
        {type = bootloader,  
         encryption = aes, keysrc=bh_blk_key, dpacm_enable, revoke_id = 0x5, aeskeyfile = ./  
plm.nky, authentication = rsa, 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}  
    }  
}
```

Replacing PLM from an Existing PDI

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

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

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

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

Example Bootgen Command to Create a PDI

Use the following command to create a PDI.

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

Replace PLM and PMC CDO in SSI technology PDIs

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

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

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

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

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

Bootgen replaces respective slr bootloader plm.elf with a new plm_v1.elf and pmcdata with pmc_data-v1.cdos.

Example Bootgen Command to Create a PDI

Use the following command to create a PDI.

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

BIF Attribute Reference

Note: If any of the below BIF attribute is used as a filename, Bootgen does not recognize it. Use absolute or relative path to bypass.

Example: `file= ./Image`

Because 'image' is a bif attribute, the path to help Bootgen differentiate is needed.

aarch32_mode

Syntax

- For AMD Zynq™ UltraScale+™ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

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

Arguments

Specified partition.

Example

- For Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

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

aeskeyfile

Syntax

- For Zynq devices and FPGAs:

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

- For Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

Description

The path to the AES keyfile. The keyfile contains the AES key used to encrypt the partitions. The contents of the key file must be written to eFUSE or BBRAM. If the key file is not present in the path specified, a new key is generated by Bootgen, which is used for encryption.

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

Arguments

Specified file name.

Return Value

None

Zynq 7000 SoC Example

The partitions `fsbl.elf` and `hello.elf` are encrypted using keys in `test.nky`.

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

Sample key (.nky) file - `test.nky`

```
Device      xc7z020c1g484;
Key 0       8177B12032A7DEEE35D0F71A7FC399027BF....D608C58;
Key StartCBC 952FD2DF1DA543C46CDDE4F811506228;
Key HMAC    123177B12032A7DEEE35D0F71A7FC3990BF....127BD89;
```

Zynq UltraScale+ MPSoC Example

Example 1:

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

```
all:
{
    [keysrc_encryption] bbram_red_key
    [bootloader, encryption=aes, aeskeyfile=test.nky] fsbl.elf
    [encryption=aes, aeskeyfile=test1.nky] hello.elf
    [encryption=aes, aeskeyfile=test2.nky] app.elf
}
```

Example 2:

Consider Bootgen creates three partitions for `hello.elf`, called `hello.elf.0`, `hello.elf.1`, and `hello.elf.2`. Sample BIF - `test_muplicte.bif`

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

Additional information:

- The partition `fsbl.elf` is encrypted with keys in `test.nky`. All `hello.elf` partitions are encrypted using keys in `test1.nky`.
- You can have unique key files for each `hello` partition by having key files named `test1.1.nky` and `test1.2.nky` in the same path as `test1.nky`.
- `hello.elf.0` uses `test1.nky`

- hello.elf.1 uses test1.1.nky
- hello.elf.2 uses test1.2.nky
- If any of the key files (test1.1.nky or test1.2.nky) is not present, Bootgen generates the key file.
- aeskeyfile format:

An .nky file accepts the following fields.

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

Note:

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

Versal Adaptive SoC Example

```
all:
{
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootloader, encryption = aes,
            keysrc = 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

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

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

Arguments

Number of bytes to be aligned.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

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

auth_params

Syntax

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

Description

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

- ppk_select: Selects which PPK to use. Options are 0 (default) or 1.
- spk_id: Specifies which SPK can be used or revoked. See [User eFUSE Support with Enhanced RSA Key Revocation](#). The default value is 0x00.

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

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

- spk_select: To differentiate spk and user efuses. Options are spk-efuse (default) and user_efuse.
- header_auth: To authenticate headers when no partition is authenticated.

Note:

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

Example

Sample BIF 1 - test.bif

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

Sample BIF 2 - test.bif

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

Sample BIF 3 - test.bif

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

Sample BIF 4 - test.bif

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

    /* FSBL - Partition-0 */
    [
        bootloader,
        destination_cpu = a53-0,
        authentication = rsa,
        spk_id = 0x3,
        spk_select = spk-efuse,
        sskfile = secondary1.pem
    ] fsbla53.elf

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

authentication

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

Description

This specifies the partition to be authenticated.

Arguments

- none: Partition not authenticated. This is the default value.
- rsa: Partition authenticated using RSA algorithm.
- ecdsa-p384 : Partition authenticated using ECDSA p384 curve
- ecdsa-p521 : Partition authenticated using ECDSA p521 curve

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

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

    image
```

```
{  
    name = pmc_subsys, id = 0x1c000001  
    partition  
    {  
        id = 0x01, type = bootloader,  
        authentication = rsa,  
        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

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

Description

To specify the binary file is in big endian format.

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

Arguments

Specified partition.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

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

bbram_kek_iv

Syntax

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

Description

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

Example

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

bh_kek_iv

Syntax

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

Description

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

Example

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

bh_keyfile

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

Description

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

Note: Obfuscated key not supported for Versal devices.

Arguments

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

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

```
all:
{
    bh_keyfile = bh_key1.txt
    bh_kek_iv = blk_iv.txt
    image
    {
        name = pmc_subsys, id = 0x1c000001
        {
            type = bootloader, encryption = aes,
            keysrc = 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

Syntax

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

Description

Initialization vector used when decrypting the black key.

Arguments

Path to file.

Example

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

bhsignature

Syntax

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

Description

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

Example

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

blocks

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

Description

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

Arguments

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

Example

- For AMD Zynq™ UltraScale+™ MPSoC:

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

- For Versal adaptive SoC:

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

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

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

boot_config

Syntax

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

Description

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

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

Options are:

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

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

- **dpacm_enable:** DPA Counter Measure Enable

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

Examples

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

boot_device

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

Note: This attribute needs to be added in BIF targeted for primary boot Image (PDI in case of Versal)

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

Arguments

Options for Zynq devices and Zynq UltraScale+ MPSoC:

- qspi32
- qspi24
- nand
- sd0

- sd1
- sd-ls
- mmc
- usb
- ethernet
- pcie
- sata

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

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

Example

Note: The following examples are for BIF for primary boot image.

- For example Zynq devices and Zynq UltraScale+ MPSoC:

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

- For example AMD Versal™ adaptive SoC:

```
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 = pmcdata, load = 0xf2000000, file =
pmc_data.cdo }
    }
    image
    {
        name = lpd, id = 0x4210002
        { id = 0x0C, type = cdo, file = lpd_data.cdo }
        { id = 0x0B, core = psm, file = psm_fw.elf }
    }
    image
    {
        name = pl_cfi, id = 0x18700000
        { id = 0x03, type = cdo, file = system.rpdo }
        { id = 0x05, type = cdo, file = system.rnpi }
    }
    image
    {
        name = fpd, id = 0x420c003
        { id = 0x08, type = cdo, file = fpd_data.cdo }
    }
}
```

bootimage

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

Arguments

Specified file name.

Example

- For FSBL:

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

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

- For `fsbl.bin` generation:

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

Use the following command:

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

- For `system.bin` generation:

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

Use the following command:

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

- For AMD Versal™ adaptive SoC:

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

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

bootloader

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

Identifies an ELF file as the FSBL or the PLM.

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

Arguments

Specified file name.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

bootvectors

Syntax

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

Description

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

Example

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

checksum

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

This specifies the partition that needs to be checksummed. This is not supported along with more secure features like [authentication](#) and [encryption](#).

Arguments

- none: No checksum operation.
- MD5: MD5 checksum operation for AMD Zynq™ 7000 SoC devices. In these devices, checksum operations are not supported for bootloaders.
- SHA3: Checksum operation for AMD Zynq™ UltraScale+™ MPSoC devices and Versal adaptive SoC.

Examples

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

copy

Syntax

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

Description

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

Example

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

core

Syntax

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

Description

This attribute specifies which core executes the partition.

Arguments

- a72-0
- a72-1
- r5-0
- r5-1
- psm

- aie
- r5-lockstep

Example

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

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

delay_auth

Syntax

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

Description

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

Example

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

delay_handoff

Syntax

```
{ delay_handoff }
```

Description

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

Example

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

delay_load

Syntax

```
{ delay_load }
```

Description

This attribute specifies that the loading of subsystem is delayed.

Example

```
test:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name=subsys_1, id=0x1c000000, delay_load
```

```
        { core=psm, file=psm.elf }
        { type=cdo, file=ps_data.cdo }
        { core=a72-0, file=a72_app.elf }
    }
```

destination_cpu

Syntax

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

Description

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

Note:

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

Arguments

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

Example

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

destination_device

Syntax

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

Description

Specifies whether the partition is targeted for PS or PL.

Arguments

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

Example

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

early_handoff

Syntax

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

Description

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

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

Example

```
all:
{
    [bootloader, destination_cpu=a53_0]fsbl.elf
    [destination_cpu=r5_0]app1.elf
    [destination_cpu=r5_1,early_handoff]app2.elf
}
```

efuse_kek_iv

Syntax

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

Description

This attribute specifies the IV that is used to encrypt the efuse black key. So, 'efuse_kek_iv' is valid with 'keysrc=efuse_blk_key'.

Example

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

efuse_user_kek0_iv

Syntax

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

Description

This attribute specifies the IV that is used to encrypt the efuse user black key0. So, 'efuse_user_kek0_iv' is valid with 'keysrc=efuse_user_blk_key0'.

Example

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

efuse_user_kek1_iv

Syntax

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

Description

This attribute specifies the IV that is used to encrypt the efuse user black key1. So, 'efuse_user_kek1_iv' is valid with 'keysrc=efuse_user_blk_key1'.

Example

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

encryption

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

Arguments

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

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

    metaheader
    {
        encryption = aes,
        keysrc = 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

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

Exception level for which the core must be configured.

Arguments

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

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

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

familykey

Syntax

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

Description

Specify family key. To obtain family key, contact an AMD representative at secure.solutions@xilinx.com.

Arguments

Path to file.

Example

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

file

Syntax

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

Description

This attribute specifies the file for creating the partition.

Example

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

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

fsbl_config

Syntax

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

Description

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

Arguments

- `bh_auth_enable`: Boot Header Authentication Enable: RSA authentication of the bootimage is done excluding the verification of PPK hash and SPK ID.
- `auth_only`: Boot image is only RSA signed. FSBL should not be decrypted. See the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) for more information.
- `opt_key`: Operational key is used for block-0 decryption. Secure Header has the opt key.
- `pufhd_bh`: PUF helper data is stored in Boot Header (default is `efuse`). PUF helper data file is passed to Bootgen using the `[puf_file]` option.
- `puf4kmode`: PUF is tuned to use in 4 k bit configuration (default is 12 k bit).
- `shutter = <value>`: 32-bit PUF_SHUT register value to configure PUF for shutter offset time and shutter open time.

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

Example

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

headersignature

Syntax

For Zynq UltraScale+ MPSoC:

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

For Versal adaptive SoC:

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

Description

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

Arguments

```
<signature_file>
```

Example

For Zynq UltraScale+ MPSoC:

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

For Versal adaptive SoC:

```
stage5:
{
    bhsignature = 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

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

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

Arguments

None

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

id

Syntax

```
id = <id>
```

Description

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

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

- partition ID - within the partition parenthesis

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

Table 47: Image IDs (Fixed for a Given Partition)

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

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

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

Example

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

image

Syntax

```
image
{
}
```

Description

This attribute is used to define a subsystem/image.

Example

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

imagestore

Syntax

```
imagestore = <id>
```

Description

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

Example

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

init

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
init = <filename>
```

Description

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

Example

A sample BIF file is shown below:

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

keysrc

Syntax

```
keysrc = <options>
```

Description

This specifies the Key source for encryption.

Arguments

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

- efuse_red_key
- efuse_blk_key
- bbram_red_key
- bbram_blk_key
- bh_blk_key

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

- user_key0
- user_key1
- user_key2
- user_key3

- user_key4
- user_key5
- user_key6
- user_key7
- efuse_user_key0
- efuse_user_blk_key0

Example

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

keysrc_encryption

Syntax

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

Description

This specifies the Key source for encryption.

Arguments

- bbram_red_key: RED key stored in BBRAM
- efuse_red_key: RED key stored in efuse
- efuse_gry_key: Grey (Obfuscated) Key stored in eFUSE.
- bh_gry_key: Grey (Obfuscated) Key stored in boot header.
- bh_blk_key: Black Key stored in boot header.

- `efuse_blk_key`: Black Key stored in eFUSE.
- `kup_key`: User Key.

Example

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

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

load

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

Sets the load address for the partition in memory.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

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

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

metaheader

Syntax

```
metaheader { }
```

Description

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

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

Example

```
test:
{
    metaheader
    {
        encryption = aes,
        keysrc = 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

Syntax

```
name = <name>
```

Description

This attribute specifies the name of the image/subsystem.

Example

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

offset

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

Arguments

Specified value and partition.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

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

optionaldata

Syntax

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

Description

This allows you to specify data ID and data file. It is also possible to specify multiple instances. Data file must be a binary file with a .bin extension. The Data IDs from 0x0 to 0x20 for Optional Data are reserved for internal use. User Optional Data ID can be anything > 0x20. For more details refer [IHT Optional Data](#).

Arguments

Filename

Optional data id

Example

```
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 = pmcdt, load = 0xf2000000, file
= pmc_data.cdo}
    }
}
```

overlay_cdo

Syntax

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

Description

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

parent_id

Syntax

```
parent_id = <id>
```

Description

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

Example

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

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

partition

Syntax

```
partition
{
```

Description

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

Example

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

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

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

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

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

partition_owner, owner

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

Arguments

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

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
all:
{
    image
    {
        { type = bootimage, file =
base.pdi }
    }
    image
    {
        name = apu_subsys, id = 0x1c000003
    }
}
```

```
        id = 0x00000000,
        core = a72-0,
        owner = non-plm,
        file = /path/to/image.ub
    }
}
```

pid

Syntax

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

Description

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

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

Example

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

pmufw_image

Syntax

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

Description

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

Arguments

Filename

Example

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

ppkfile

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
ppkfile = <filename>
```

Description

The PPK key is used to authenticate partitions in the boot image.

See [Using Authentication](#).

Arguments

Specified file name.

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

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

presign

Syntax

For Zynq 7000 and Zynq UltraScale+ MPSoC devices:

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

For Versal adaptive SoC:

```
presign = <signature_file>
```

Description

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

- **<signature_file>**: Specifies the signature file.
- **<partition>**: Lists the partition to which the **<signature_file>** is applied.

Example

For Zynq 7000 and Zynq UltraScale+ MPSoC devices:

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

pskfile

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
pskfile = <filename>
```

Description

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

Arguments

Specified file name.

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

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

puf_file

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For Versal adaptive SoC:

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

Description

PUF helper data file.

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

See [Black/PUF Keys](#) for more information.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

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

reserve

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

Description

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

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

Arguments

Specified partition

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

```
        }
    image
    {
        name = apu_ss, id = 0x1c000000
        { reserve = 0x1000, file = data.bin }
    }
}
```

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

split

Syntax

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

Description

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

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

Normal mode splits as follows:

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

Slaveboot is supported only for Zynq UltraScale+ MPSoC, and normal is supported for both Zynq 7000 and Zynq UltraScale+ MPSoC. Along with the split mode, output format can also be specified as `bin` or `mcs`.

Options

The available options for argument mode are:

- slaveboot
- normal
- bin
- mcs

Example

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

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

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

spkfile

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

```
spkfile = <filename>
```

Description

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

Arguments

Specified file name.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

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

- For AMD Versal™ adaptive SoC:

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

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

spksignature

Syntax

For Zynq and Zynq UltraScale+ MPSoC devices:

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

For Versal adaptive SoC:

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

Description

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

Arguments

Specified file name.

Example

For Zynq and Zynq UltraScale+ MPSoC devices:

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

For Versal adaptive SoC:

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

spk_select

Syntax

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

or

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

Description

Options are:

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

Partitions loaded by CSU ROM is always use spk_efuse.

Note: The `spk_id` eFUSE specifies which key is valid. Hence, the ROM checks the entire field of `spk_id` eFUSE against the SPK ID to make sure its a bit for bit match.

The user eFUSE specifies which key ID is *not* valid (has been revoked). Hence, the firmware (non-ROM) checks to see if a given user eFUSE that represents the SPK ID has been programmed.

`spk_select = user-efuse` indicates that user eFUSE is used for that partition.

Example

```
the_ROM_image:
{
    [auth_params]ppk_select = 0
    [pskfile]psk.pem
    [sskfile]ssk1.pem

    [
        bootloader,
        authentication = rsa,
        spk_select = spk-efuse,
        spk_id = 0x5,
        sskfile = ssk2.pem
    ] zynqmp_fsbl.elf

    [
        destination_cpu = a53-0,
        authentication = rsa,
        spk_select = user-efuse,
        spk_id = 0xF,
        sskfile = ssk3.pem
    ] application1.elf

    [
        destination_cpu = a53-0,
        authentication = rsa,
        spk_select = spk-efuse,
        spk_id = 0x6,
        sskfile = ssk4.pem
    ] application2.elf
}
```

sskfile

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[sskfile] <key filename>
```

- For AMD Versal™ adaptive SoC:

```
sskfile = <filename>
```

Description

The SSK is used to authenticate partitions in the boot image. For more information, see [Using Authentication](#).

Arguments

Specified file name.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
all:
{
    [pskfile] primarykey.pem
    [sskfile] secondarykey.pem
    [bootloader, authentication=rsa]fsbl.elf
    [authentication=rsa] hello.elf
}
```

- For AMD Versal™ adaptive SoC:

```
all:
{
    boot_config {bh_auth_enable}
    image
    {
        name = pmc_ss, id = 0x1c000001
        { type=bootloader, authentication=rsa, file=plm.elf,
        pskfile=primary0.pem, sskfile=secondary0.pem }
        { type = pmcdt, load = 0xf2000000, file=pmc_cdo.bin }
        { type=cdo, authentication=rsa, file=fpd_cdo.bin, pskfile =
        primary1.pem, sskfile = secondary1.pem }
    }
}
```

Note: The secret key file contains the public key component of the key. You need not specify the PPK when the PSK is mentioned.

startup

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[startup = <value>] <filename>
```

- For AMD Versal™ adaptive SoC:

```
{ startup = <value>, file = <filename> }
```

Description

This option sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute. This is valid only for binary partitions.

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
all:
{
    [bootloader] fsbl.elf
    [startup=0x1000000] app.bin
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { core=a72-0, load=0x1000, startup = 0x1000, file = apu.bin }
    }
}
```

Note: *base.pdi is the PDI generated by Vivado.

trustzone

Syntax

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
[trustzone = <options> ] <filename>
```

- For AMD Versal™ adaptive SoC:

```
{ trustzone = <options>, file = <filename> }
```

Description

Configures the core to be TrustZone secure or non-secure. Options are:

- secure
- nonsecure (default)

Example

- For Zynq devices and Zynq UltraScale+ MPSoC:

```
all:
{
    [bootloader, destination_cpu=a53-0] fsbl.elf
    [exception_level=el-3, trustzone = secure] bl31.elf
}
```

- For AMD Versal™ adaptive SoC:

```
new_bif:
{
    image
    {
        { type = bootimage, file = base.pdi }
    }
    image
    {
        name = apu_ss, id = 0x1c000000
        { load = 0x1000, file = system.dtb }
        { exception_level = el-2, file = u-boot.elf }
        { core = a72-0, exception_level = el-3, trustzone, file =
    bl31.elf }
    }
}
```

Note: *base.pdi is the PDI generated by Vivado.

type

Syntax

```
{ type = <options> }
```

Description

This attribute specifies the type of partition. The options are as follows.

- bootloader
- pmcdata
- cdo
- bootimage

Example

```
new_bif:  
{  
    image  
    {  
        { type = bootimage, file = base.pdi }  
    }  
    image  
    {  
        name = apu_ss, id = 0x1c000000  
        { core = a72-0, file = apu.elf }  
    }  
}
```

Note: *base.pdi is the PDI generated by Vivado.

udf_bh

Syntax

```
[udf_bh] <filename>
```

Description

Imports a file of data to be copied to the user defined field (UDF) of the Boot Header. The input user defined data is provided through a text file in the form of a hex string. Total number of bytes in UDF in AMD SoCs:

- zynq: 76 bytes
- zynqmp: 40 bytes

Arguments

Specified file name.

Example

```
all:  
{  
    [udf_bh]test.txt  
    [bootloader]fsbl.elf  
    hello.elf  
}
```

The following is an example of the input file for udf_bh:

Sample input file for `udf_bh` - `test.txt`

```
123456789abcdef85072696e636530300301440408706d616c6c6164000508
266431530102030405060708090a0b0c0d0e0f101112131415161718191a1b
1c1d1
```

udf_data

Syntax

```
[udf_data=<filename>] <partition>
```

Description

Imports a file containing up to 56 bytes of data into user defined field (UDF) of the Authentication Certificate. For more information, see [Authentication](#) for more information about authentication certificates.

Arguments

Specified file name.

Example

```
all:
{
    [pskfile] primary0.pem
    [sskfile]secondary0.pem
    [bootloader, destination_cpu=a53-0,
    authentication=rsa, udf_data=udf.txt]fsbl.elf
        [destination_cpu=a53-0, authentication=rsa] hello.elf
}
```

userkeys

Syntax

```
userkeys = <filename>
```

File Format

```
user_key0 <userkey0 value>
user_key1 <userkey1 value>
user_key2 <userkey2 value>
user_key3 <userkey3 value>
user_key4 <userkey4 value>
user_key5 <userkey5 value>
user_key6 <userkey6 value>
user_key7 <userkey7 value>
```

Description

The path to the user keyfile. The keyfile contains user keys used to encrypt the partitions. The size of user key can be 128 or 256 bits. The 128-bit key can be used only for run-time loaded partitions.

Example

In the following example, FPD partition uses the key source as `user_key2`, so the `.nky` file for this partition must have the `user_key2` from the `userkeys` file as the `key0`. This `key0` from the `.nky` file is then used by Bootgen for encryption. The PLM uses the `user_key2` programmed by `pmc_data` during decryption.

```
new_bif:
{
  userkeys = userkeyfile.txt
  id_code = 0x14ca8093
  extended_id_code = 0x01
  id = 0x2
  image
  {
    name = pmc_subsys
    id = 0x1c000001
    partition
    {
      id = 0x01
      type = bootloader
      encryption = aes
      keysrc=bbram_red_key
      aeskeyfile = inputs/keys/enc/bbram_red_key.nky
      dpacm_enable
      file = gen_files/plm.elf
    }
    partition
    {
      id = 0x09
      type = pmcdata, load = 0xf2000000
      file = gen_files/pmc_data.cdo
    }
  }
  image
  {
    name = lpd
    id = 0x4210002
    partition
    {
```

```
    id = 0x0C
    type = cdo
    file = gen_files/lpd_data.cdo
}
partition
{
    id = 0x0B
    core = psm
    file = static_files/psm_fw.elf
}
}
image
{
    name = pl_cfi
    id = 0x18700000
partition
{
    id = 0x03
    type = cdo
    file = design_1_wrapper.rcdo
}
partition
{
    id = 0x05
    type = cdo
    file = design_1_wrapper.rnpi
}
}
image
{
    name = fpd
    id = 0x420c003
partition
{
    id = 0x08
    type = cdo
    file = gen_files/fpd_data.cdo
    encryption = aes
    keysrc=user_key2
    aeskeyfile = userkey2.nky
}
}
image
{
    name = ss_apu
    id = 0x1c000000
partition
{
    id = 0x61
    core = a72-0
    file = ./wrk_a72_r5/perip_a72/Debug/perip_a72.elf
}
}
```

xip_mode

Syntax

```
[xip_mode] <partition>
```

Description

Indicates 'eXecute In Place' for FSBL to be executed directly from QSPI flash.

Note: This attribute is only applicable for an FSBL/Bootloader partition.

Arguments

Specified partition.

Example

This example shows how to create a boot image that executes in place for an AMD Zynq™ UltraScale+™ MPSoC device.

```
all:
{
    [bootloader, xip_mode] fsbl.elf
    application.elf
}
```

Command Reference

See [Commands and Descriptions](#) for the device families supported by each of these commands.

arch

Syntax

```
-arch [options]
```

Description

AMD family architecture for which the boot image needs to be created.

Arguments

- zynq: AMD Zynq™ 7000 device architecture. This is the default value of the family architecture for which the boot image needs to be created.
- zynqmp: AMD Zynq™ UltraScale+™ MPSoC architecture.
- fpga: Image is targeted for other FPGA architectures.
- versal: This image is targeted for AMD Versal™ devices.

Return Value

None

Example

```
bootgen -arch zynq -image test.bif -o boot.bin
```

authenticatedjtag

Syntax

```
-authenticatedjtag [options] [filename]
```

Description

Used to enable JTAG during secure boot.

Arguments

- rsa
- ecdsa

Example

```
bootgen -arch versal -image boot.bif -w -o boot.bin -authenticatedjtag rsa  
authJtag-rsa.bin
```

bif_help

Syntax

```
bootgen -bif_help
```

```
bootgen -bif_help aeskeyfile
```

Description

Lists the supported BIF file attributes. For a more detailed explanation of each bif attribute, specify the attribute name as argument to `-bif_help` on the command line.

dual_ospি_mode

Syntax

```
bootgen -arch versal -dual_ospি_mode stacked <size>
```

Description

Generates two output files for dual OSPI stacked configuration, size (in MB) of the flash needs to be mentioned (64, 128, or 256).

Example

This example generates two output files for independently programming to both flashes in a OSPI dual stacked configuration. The first 64 MB of the actual image is written to first file and the remainder to the second file. In case the actual image itself is less than 64 MB, only one file is generated. This is only supported for Versal adaptive SoC.

```
bootgen -arch versal -image test.bif -o -boot.bin -dual_ospি_mode stacked 64
```

Arguments

- stacked, <size>

dual_qspi_mode

Syntax

```
bootgen -dual_qspi_mode [parallel] | [stacked <size>]
```

Description

Generates two output files for dual QSPI configurations. In the case of stacked configuration, size (in MB) of the flash needs to be mentioned (16, 32, 64, 128, or 256).

Examples

This example generates two output files for independently programming to both flashes in QSPI dual parallel configuration.

```
bootgen -image test.bif -o -boot.bin -dual_qspi_mode parallel
```

This example generates two output files for independently programming to both flashes in a QSPI dual stacked configuration. The first 64 MB of the actual image is written to first file and the remainder to the second file. In case the actual image itself is less than 64 MB, only one file is generated.

```
bootgen -image test.bif -o -boot.bin -dual_qspi_mode stacked 64
```

Arguments

- parallel
 - stacked <size>
-

dump

Syntax

```
-dump [options]
```

Description

This command dumps the contents of boot header in to a separate binary file while generating PDI.

Example

```
[bootgen -image test.bif -o -boot.bin -log trace -dump bh]
```

Arguments

- empty: Dumps the partitions as binary files.
- bh: Dumps boot header as a separate file.

Note: Boot header is dumped as a separate binary file along with PDI. PDI generated is not be stripped of the boot header, but it retains the boot header.

dump_dir

Syntax

```
dump_dir <path>
```

Description

This option is used to specify a directory location to write the contents of -dump command.

Example

```
bootgen -arch versal -dump boot.bin -dump_dir <path>
```

efuseppkbits

Syntax

```
bootgen -image test.bif -o boot.bin -efuseppkbits efusefile.txt
```

Arguments

efusefile.txt

Description

This option specifies the name of the eFUSE file to be written to contain the PPK hash. This option generates a direct hash without any padding. The `efusefile.txt` file is generated containing the hash of the PPK key, where:

- AMD Zynq™ 7000 uses the `SHA2` protocol for hashing.
 - AMD Zynq™ UltraScale+™ MPSoC and Versal adaptive SoC uses the `SHA3` for hashing.
-

enable_auth_opt

Syntax

```
bootgen -arch versal -image test.bif -o -boot.bin -enable_auth_opt
```

Description

This option is used to enable authentication optimization. Boot image generated has meta header and partition hashes stored as part of optional data.

Note: This option is supported only for AMD Versal™ SoC.

encrypt

Syntax

```
bootgen -image test.bif -o boot.bin -encrypt <efuse|bbram|>
```

Description

This option specifies how to perform encryption and where the keys are stored. The NKY key file is passed through the BIF file attribute `aeskeyfile`. Only the source is specified using command line.

Arguments

Key source arguments:

- `efuse`: The AES key is stored in eFUSE. This is the default value.
- `bbram`: The AES key is stored in BBRAM.

encryption_dump

Syntax

```
bootgen -arch zynqmp -image test.bif -encryption_dump
```

Description

Generates an encryption log file, `aes_log.txt`. The `aes_log.txt` generated has the details of AES Key/IV pairs used for encrypting each block of data. It also logs the partition and the AES key file used to encrypt it.

Note: This option is supported only for AMD Zynq™ UltraScale+™ MPSoC.

Example

```
all:
{
    [bootloader, encryption=aes, aeskeyfile=test.nky] fsbl.elf
    [encryption=aes, aeskeyfile=test1.nky] hello.elf
}
```

fill

Syntax

```
bootgen -arch zynq -image test.bif -fill 0xAB -o boot.bin
```

Description

This option specifies the byte to use for filling padded/reserved memory in <hex byte> format.

Outputs

The `boot.bin` file in the `0xAB` byte.

Example

The output image is generated with name `boot.bin`. The format of the output image is determined based on the file extension of the file given with `-o` option, where `-fill`: Specifies the Byte to be padded. The <hex byte> is padded in the header tables instead of `0xFF`.

```
bootgen -arch zynq -image test.bif -fill 0xAB -o boot.bin
```

generate_hashes

Syntax

```
bootgen -image test.bif -generate_hashes
```

Description

This option generates hash files for all the partitions and other components to be signed like boot header, image, and partition headers. This option generates a file containing PKCS#1v1.5 padded hash for the AMD Zynq™ 7000 format:

Table 48: Zynq: SHA-2 (256-bytes)

Value	SHA-2 Hash	T-Padding	0x0	0xFF	0x01	0x00
Number of bytes	32	19	1	202	1	1

This option generates the file containing PKCS#1v1.5 padded hash for the AMD Zynq™ UltraScale+™ MPSoC format:

Table 49: ZynqMP: SHA-3 (384-bytes)

Value	0x0	0x1	0xFF	0xFF	T-Padding	SHA-3 Hash
Number of bytes	1	1	314	1	19	48

Example

```
test:  
{  
    [pskfile] ppk.txt  
    [sskfile] spk.txt  
    [bootloader, authentication=rsa] fsbl.elf  
    [authentication=rsa] hello.elf  
}
```

Bootgen generates the following hash files with the specified BIF:

- bootheader hash
- spk hash
- header table hash
- fsbl.elf partition hash
- hello.elf partition hash

generate_keys

Syntax

```
bootgen -image test.bif -generate_keys <rsa|pem|obfuscated>
```

Description

This option generates keys for authentication and obfuscated key used for encryption.

Note: For more information on generating encryption keys, see [Key Generation](#).

Authentication Key Generation Example

Authentication key generation example. This example generates the authentication keys in the paths specified in the BIF file.

Examples

```
image:  
{  
    [ppkfile] <path/ppkgenfile.txt>  
    [pskfile] <path/pskgenfile.txt>  
    [spkfile] <path/spkgenfile.txt>  
    [sskfile] <path/sskgenfile.txt>  
}
```

Obfuscated Key Generation Example

This example generates the obfuscated in the same path as that of the `familykey.txt`.

Command:

```
bootgen -image test.bif -generata_keys rsa
```

The Sample BIF file is shown in the following example:

```
image:
{
    [aeskeyfile] aes.nky
    [bh_key_iv] bhkeyiv.txt
    [familykey] familykey.txt
}
```

Arguments

- rsa
- pem
- obfuscated

h, help

Syntax

```
bootgen -help
bootgen -help arch
```

Description

Lists the supported command line attributes. For a more detailed explanation of each attribute, specify the attribute name as argument to `-help` on the command line.

image

Syntax

```
-image <BIF_filename>
```

Description

This option specifies the input BIF file name. The BIF file specifies each component of the boot image in the order of boot and allows optional attributes to be specified to each image component. Each image component is usually mapped to a partition, but in some cases an image component can be mapped to more than one partition if the image component is not contiguous in memory.

Arguments

bif_filename

Example

```
bootgen -arch zynq -image test.bif -o boot.bin
```

The Sample BIF file is shown in the following example:

```
the_ROM_image :  
{  
    [init] init_data.int  
    [bootloader] fsbl.elf  
    Partition1.bit  
    Partition2.elf  
}
```

log

Syntax

```
bootgen -image test.bif -o -boot.bin -log trace
```

Description

Generates a log while generating the boot image. There are various options for choosing the level of information. The information is displayed on the Console and in the log file, named `bootgen_log.txt` is generated in the current working directory.

Arguments

- error: Only the error information is captured.
- warning: The warnings and error information is captured. This is the default value.
- info: The general information and all the above information is captured.
- trace: More detailed information is captured along with the information above.

nonbooting

Syntax

```
bootgen -arch zynq -image test.bif -o test.bin -nonbooting
```

Description

This option is used to create an intermediate boot image. An intermediate `test.bin` image is generated as output even in the absence of secret key, which is required to generate an authenticated image. This intermediate image cannot be booted.

Example

```
all:
{
    [ppkfile]primary.pub
    [spkfile]secondary.pub
    [spksignature]secondary.pub.sha256.sig

    [bootimage, authentication=rsa, presign=fsbl_0.elf.0.sha256.sig]fsbl_e.bin
}
```

O

Syntax

```
bootgen -arch zynq -image test.bif -o boot.<bin|mcs>
```

Description

This option specifies the name of the output image file with a `.bin` or `.mcs` extension.

Outputs

A full boot image file in either BIN or MCS format.

Example

```
bootgen -arch zynq -image test.bif -o boot.mcs
```

The boot image is output in an MCS format.

p

Syntax

```
bootgen -image test.bif -o boot.bin -p xc7z020clg48 -encrypt efuse
```

Description

This option specifies the partname of the AMD device. This is needed for generating an encryption key. It is copied verbatim to the *.nky file in the Device line of the nky file. This is applicable only when encryption is enabled. If the key file is not present in the path specified in BIF file, then a new encryption key is generated in the same path and `xc7z020clg484` is copied along side the `Device` field in the nky file. The generated image is an encrypted image.

padimageheader

Syntax

```
bootgen -image test.bif -w on -o boot.bin -padimageheader <0|1>
```

Description

This option pads the Image Header Table and Partition Header Table to maximum partitions allowed, to force alignment of following the partitions. This feature is enabled by default. Specifying a 0 disables this feature. The `boot.bin` has the image header tables and partition header tables in actual and no extra tables are padded. If nothing is specified or if `-padimageheader=1`, the total image header tables and partition header tables are padded to max partitions.

Arguments

- 1: Pad the header tables to max partitions. This is the default value.
- 0: Do not pad the header tables.

Image or Partition Header Lengths

- For Zynq devices, the maximum partition is 14.
- For Zynq UltraScale+ MPSoCs, the maximum partition is 32.

process_bitstream

Syntax

```
-process_bitstream <bin|mcs>
```

Description

Processes only the bitstream from the BIF and outputs it as an MCS or a BIN file. For example: If encryption is selected for bitstream in the BIF file, the output is an encrypted bitstream.

Arguments

- bin: Output in BIN format.
- mcs: Output in MCS format.

Returns

Output generated is bitstream in BIN or MCS format; a processed file without any headers attached.

read

Syntax

```
-read <filename> [options]
```

Description

Used to read boot headers, image headers, and partition headers based on the options.

Arguments

- bh: To read boot header from boot image in human readable form
- iht: To read image header table from boot image
- ih: To read image headers from boot image.
- pht: To read partition headers from boot image
- ac: To read authentication certificates from boot image

Example

```
bootgen -arch zynqmp -read BOOT.bin
```

Note: -read is not supported for bootimages/PDIs in MCS format

spksignature

Syntax

```
bootgen -image test.bif -w on -o boot.bin -spksignature spksignfile.txt
```

Description

This option is used to generate the SPK signature file. This option must be used only when `spkfile` and `pskfile` are specified in BIF. The SPK signature file (`spksignfile.txt`) is generated.

Option

Specifies the name of the signature file to be generated.

split

Syntax

```
bootgen -arch zynq -image test.bif -split bin
```

Description

This option outputs each data partition with headers as a new file in MCS or BIN format.

Outputs

Output files generated are:

- Bootheader + Image Headers + Partition Headers + Fsbl.elf
- Partition1.bit
- Partition2.elf

Example

```
the_ROM_image:  
{  
    [bootloader] Fsbl.elf  
    Partition1.bit  
    Partition2.elf  
}
```

verify

Syntax

```
bootgen -arch zynqmp -verify boot.bin
```

Description

This option is used for verifying authentication of a boot image. All the authentication certificates in a boot image are verified against the available partitions. Verification is performed in the following steps:

1. Verify header authentication certificate:
 - For Zynq UltraScale+ MPSoC: verify SPK signature and verify header signature.
 - For Versal: verify SPK signature, verify IHT signature, and verify meta header signature.
2. Verify bootloader authentication certificate: verify boot header signature, verify SPK signature, and verify bootloader signature.
3. Verify partition authentication certificate: verify SPK signature and verify partition signature.

This is repeated for all partitions in the given boot image.

verify_kdf

Syntax

```
bootgen -arch zynqmp -verify_kdf testVec.txt
```

Description

The format of the `testVec.txt` file is as below.

```
L = 256
KI = d54b6fd94f7cf98fd955517f937e9927f9536caeb148fba1818c1ba46bba3a4
FixedInputDataByteLen = 60
FixedInputData =
94c4a0c69526196c1377cebf0a2ae0fb4b57797c61bea8eeb0518ca08652d14a5e1bd1b116b1
794ac8a476acbdbbcd4f6142d7b8515bad09ec72f7af
```

Bootgen uses the Counter Mode KDF to generate the output key (KO) based on the given input data in the test vector file. This KO is printed on the console for you to compare.

W

Syntax

```
bootgen -image test.bif -w on -o boot.bin
or
bootgen -image test.bif -w -o boot.bin
```

Description

This option specifies whether to overwrite an existing file or not. If the file `boot.bin` already exists in the path, then it is overwritten. Options `-w on` and `-w` are treated as same. If the `-w` option is not specified, the file is be overwritten by default.

Arguments

- `on`: Specified with the `-w on` command with or `-w` with no argument. This is the default value.
 - `off`: Specifies to not overwrite an existing file.
-

zynqmpes1

Syntax

```
bootgen -arch zynqmp -image test.bif -o boot.bin -zynqmpes1
```

Description

This option specifies that the image generated is used on ES1 (1.0). This option makes a difference only when generating an Authenticated image; otherwise, it is ignored. The default padding scheme is for (2.0) ES2 and above.

Initialization Pairs and INT File Attribute

Initialization pairs let you easily initialize processing systems (PS) registers for the MIO multiplexer and flash clocks. This allows the MIO multiplexer to be fully configured before the FSBL image is copied into OCM or executed from flash with eXecute in place (XIP), and allows for flash device clocks to be set to maximum bandwidth speeds.

There are 256 initialization pairs at the end of the fixed portion of the boot image header. Initialization pairs are designated as such because a pair consists of a 32-bit address value and a 32-bit data value. When no initialization is to take place, all of the address values contain 0xFFFFFFFF, and the data values contain 0x00000000. Set initialization pairs with a text file that has an .int file extension by default, but can have any file extension.

The [init] file attribute precedes the file name to identify it as the INT file in the BIF file. The data format consists of an operation directive followed by:

- An address value
- An = character
- A data value

The line is terminated with a semicolon (;). This is one .set. operation directive; for example:

```
.set. 0xE0000018 = 0x00000411; // This is the 9600 uart setting.
```

Bootgen fills the boot header initialization from the INT file up to the 256 pair limit. When the BootROM runs, it looks at the address value. If it is not 0xFFFFFFFF, the BootROM uses the next 32-bit value following the address value to write the value of address. The BootROM loops through the initialization pairs, setting values, until it encounters a 0xFFFFFFFF address, or it reaches the 25sixth initialization pair.

Bootgen provides a full expression evaluator (including nested parenthesis to enforce precedence) with the following operators:

```
* = multiply/  
= divide  
% = mod  
an address value  
ulo divide  
+ = addition
```

```
- = subtraction
~ = negation
>> = shift right
<< = shift left
& = binary and
| = binary or
^ = binary nor
```

The numbers can be hex (0x), octal (0o), or decimal digits. Number expressions are maintained as 128-bit fixed-point integers. You can add white space around any of the expression operators for readability.

CDO Utility

The CDO utility (cdoutil) is a program that allows to process CDO files in various ways. CDO files are binary files created in the AMD Vivado™ Design Suite for AMD Versal™ devices based on user configuration for clocks, PLLs, and MIO. CDOs are part of the PDI, and are loaded/executed by the PLM. For AMD Zynq™ 7000 devices and AMD Zynq™ UltraScale+™ MPSoCs, the configuration is part of `ps7/psu_init.c/h` files, which are compiled along with the FSBL.

Accessing

The cdoutil is available as part of the Vivado Design Suite/AMD Vitis™ unified software platform/Bootgen installation at `<INSTALL_DIR>/bin/cdoutil`.

Usage

The general command line syntax for cdoutil is:

```
cdoutil <options> <input(s)>
```

The default function of cdoutil is to decode the input file and print out the CDO.

Command Line Options

There are a number of options to change the default behavior:

Table 50: Command Line Options

Option	Description
<code>-address-filter-file <path></code>	Specify address filter file
<code>-annotate</code>	Annotate source output with details of commands
<code>-device <type></code>	Specify device name, default is xcvc1902
<code>-help</code>	Print help information
<code>-output-binary-be</code>	Output CDO commands in big endian binary format
<code>-output-binary-le</code>	Output CDO commands in little endian binary format

Table 50: Command Line Options (cont'd)

Option	Description
<code>-output-file <path></code>	Specify output file, default is stdout
<code>-output-modules</code>	Output list of modules used by input file(s)
<code>-output-raw-be</code>	Output CDO commands in big endian raw format
<code>-output-raw-le</code>	Output CDO commands in little endian raw format
<code>-output-source</code>	Output CDO commands in source format (default)
<code>-remove-comments</code>	Remove comments from input
<code>-rewrite-block</code>	Rewrite block write commands to multiple write commands
<code>-rewrite-sequential</code>	Rewrite sequential write commands to a single block write command
<code>-verbose</code>	Print log information
<code>post-process <mode></code>	Post process PMCFW commands to PLM commands
<code>cfu-stream-keyhole-size <size></code>	Override default CFU stream keyhole size
<code>random-commands <count></code>	Generate <count> random commands
<code>apropos <keywords></code>	Search device register information for <keywords> and output matches
<code>overlay <path></code>	Specify overlay file

Note: `-output-raw-be` is preferred as the Vivado Design Suite produces CDOs in big endian raw format. `-output-raw-le`, `-output-binary-be`, and `-output-binary-le` are not preferred options.

Address Filter File

The address filter file is specified using the `-address-filter-file <path>`. The purpose of this file is to specify modules that must be removed from the configuration. The address filter file is a text file where each line starting with the dash (minus) character specifies an address range for which all initializations should be removed. Example:

```
# Remove configuration of UART0
-UART0
```

The list of modules used in a design can be generated using the `-output-modules` option. This can be a useful starting point for the address filter file.

Examples

Converting Binary to Source without Annotations

```
cdoutil -output-file test.txt test.bin
```

Example output:

```
version 2.0
write 0xfc50000 0
write 0xfc50010 0
write 0xfc50018 0x1
write 0xfc5001c 0
write 0xfc50020 0
write 0xfc50024 0xffffffff
```

Converting Binary to Source with Annotations

```
cdoutil -annotate -output-file test.txt test.bin
```

Example output:

```
version 2.0
# PCIEA_ATTRIB_0.MISC_CTRL.slverr_enable[0]=0x0
write 0xfc50000 0
# PCIEA_ATTRIB_0.ISR.{dpll_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x0}
write 0xfc50010 0
# PCIEA_ATTRIB_0.IER.{dpll_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x1}
write 0xfc50018 0x1
# PCIEA_ATTRIB_0.IDR.{dpll_lock_timeout_err[1]=0x0, addr_decode_err[0]=0x0}
write 0xfc5001c 0
# PCIEA_ATTRIB_0.ECO_0.eco_0[31:0]=0x0
write 0xfc50020 0
# PCIEA_ATTRIB_0.ECO_1.eco_1[31:0]=0xffffffff
write 0xfc50024 0xffffffff
```

Editing Binary CDO File

```
cdoutil -annotate -output-file test.txt test.bin
vim text.txt
cdoutil -output-binary-be -output-file test-new.bin test.txt
```

Make sure .bif file is using test-new.bin instead of test.bin, then rerun bootgen to create the .pdi file.

Converting Source to Binary

```
cdoutil -output-binary-be -output-file test.bin test.txt
```

Design Advisories for Bootgen

- AMD recommends that you generate your own keys for fielded systems and then provide those keys to the development tools. See [Answer Record 76171](#) for more information.
- In this release, few encryption key rolling blocks are supported for Versal Adaptive SoC. See [Answer Record 76515](#) for more information.
- Versal adaptive SoC 2022.2 onwards, to reduce the size of PLM and ensure it fits in the PPU RAM, the maximum number of partitions allowed is reduced from 32 to 20, and the maximum number of images/sub systems allowed is reduced from 32 to 10. If the limit exceeds, Bootgen errors out while creating the Boot Image.

The option for disabling this error is to use the `BOOTGEN_SKIP_MAX_PARTITIONS_CHECK` environment variable.

The user needs to ensure to handle changes in the PLM code as well, and then proceed to create PDIs with any number of partitions/images.

- Tandem/Partial bitstream processing is not supported: Refer to [Answer Record 35054](#) for more details.
- AMD Versal™ Bootgen Support Updates: Refer to [Answer Record 34634](#) for more details.

Vitis Python CLI

Graphical development environments such as the Vitis IDE are useful for getting up to speed on development for a new processor architecture. It helps to abstract away and group most of the common functions into logical wizards that even a novice can use. However, scriptability of a tool is also essential for providing the flexibility to extend what is done with that tool. It is particularly useful when developing regression tests that be run nightly or when running a set of commands that are frequently used.

The Vitis command line interface (CLI) is an interactive and scriptable command-line interface to the Vitis IDE. As with other AMD tools, the scripting language for Vitis CLI is based on the Python. You can run Vitis CLI commands interactively or script the commands for automation.

Vitis CLI supports Vitis project management, configuration, building, and debugging, such as:

- Creating platform projects and domains
- Creating system and application projects
- Configuring and building domains/BSPs and applications
- Managing repositories
- Downloading and running applications on hardware targets
- Reading and writing registers
- Setting break points and watch expressions

Python API: A command-line tool for creating and managing projects in Vitis

The Vitis IDE provides logical wizards within the development environment to help you develop your design. This is useful for developing a new processor architecture, particularly for novice users. For more flexibility and extended functionality, however, it is essential to become familiar with using scripts in the tool. Scripts are especially useful for developing regression tests that will be run at regular intervals or when running a frequently used set of commands. The AMD Vitis™ command line interface (CLI) is an interactive and scriptable command-line interface to the AMD Vitis™ IDE. As with other AMD tools, the scripting language for AMD Vitis™ CLI is based on the Python. You can run AMD Vitis™ CLI commands interactively or script the commands for automation. AMD Vitis™ CLI supports AMD Vitis™ project management, configuration, building and debugging, such as:

- Creating platform projects and domains
- Creating system and application projects
- Configuring and building domains/BSPs and application
- Managing repositories
- Downloading and running applications on hardware targets
- Reading and writing registers
- Setting break points and watch expressions

To execute Python APIs, first, you need to establish the connection between the server and the client. Python APIs can be executed in command line mode or as a Python script. The AMD Vitis™ Unified IDE supports two ways to run Python APIs:

1. Run Python API in CLI (Command-Line Interface) mode: Executing the Python API in CLI mode is supported in interactive mode only. For more details, see "Vitis Interactive Python Shell" in *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#))

```
vitis -i (This opens the Vitis interactive shell)
```

2. Run Python API in a Python Script: The Vitis Unified IDE supports the execution of Python script in batch mode as well as in an interactive mode.

In batch mode:

```
In batch mode:  
$ vitis -s <.py>  
In Interactive mode:  
$ vitis -i  
(This opens the Python interactive shell)  
vitis [1]: run <.py>
```

Some Python APIs are helpful in managing client life cycles, create Vitis workspaces and manage Vitis project flows. These APIs are described in the following table. Refer the Vitis API CLI documentation for more details.

Table 51: Python APIs: Manage client

Python API	Description	Example
create_client	Creates a client instance.	client=vitis.create.client()
dispose	Closes all client connections and terminates the connection to the server.	client.dispose()
exit	Closes the session.	exit()

For more details on all the Python APIs supported by Vitis, refer to
`<vitis_installation_path>/cli/api_docs/build/html/vitis.html`

Python examples are provided with the installation for reference:
`<vitis_installation_path>/cli/examples`

After creating and building the components and system project, the workspace can be directly opened in the Vitis IDE using the command below:

```
vitis -w <workspace_path>
```

Components and system projects are opened in the Vitis IDE with build and created status=done, if they are created and build through Python APIs.

The Python APIs used for component creation and build are explained in the following sections.

Managing Vitis IDE Components through Python APIs

Before running the script, you must set up the environment variables for the Vitis Unified IDE. Refer to "Setting Up the Vitis Tool Environment" in *Vitis Unified Software Platform Documentation: Application Acceleration Development* ([UG1393](#)) to set up the Vitis Unified IDE environment. The Vitis IDE supports Python APIs for project flow management, creation and building of the following Vitis components:

- Platform Control
- Application Component
- System Project

System Project

The Vitis Unified IDE supports Python APIs to create and build a system project. After creating platform component and application component, the system project is created. The application components then can be added to the system project and built. Host components can also be built individually. However, when system project build is triggered using Python APIs, it builds all the components associated and the system.

Table 52: Python APIs: System Project

Python API	Description	Python API Example
create_sys_project	Creates a system project for the given template. ¹	proj = client.create_sys_project(name='system_project', platform=<platform_path>)
add_component	Adds the specified component to the given system project	proj.add_component(name='aie_component', container_name=['system_prj_lab1']) proj.add_component(name='hls_component', container_name=['system_prj_lab1'])
build	Initiates the build of a system project for the given build target.	proj.build(target='hw')

Notes:

1. Accelerated flows are not supported for Embedded installer.

The Python script to create a system project and build for target=Hardware to export Vitis metadata archive can be seen as:

```
# Add package: Vitis Python CLI
import vitis

# Create a Vitis client object
client = vitis.create_client()

# Set Vitis Workspace
client.set_workspace(path=".//workspace")

# Defining names for platform, application_component and e
plat_name="vck190_embd"
comp_name="standalone_embd_app"
```

```
# Create and build platform component for vck190 for
standalone_psv_cortexa72
platform_obj=client.create_platform_component(name=plat_name,
hw_design="vck190", cpu="psv_cortexa72_0", os="standalone")
platform_obj.build()

# This returns the platform xpfm path
platform_xpfm=client.find_platform_in_repos(plat_name)

# Create and build application component
comp = client.create_app_component(name=comp_name, platform =
platform_xpfm, domain = "standalone_psv_cortexa72_0", template =
"hello_world")
comp.build()

# Create system project
sys_proj = client.create_sys_project(name="system_project",
platform=platform_xpfm, template="empty_accelerated_application")
# Add application component to the system project
sys_proj_comp = sys_proj.add_component(name="hello_world")
# Build system project
sys_proj_comp.build()

vitis.dispose()
```

Conclusion

Apart from the discussed examples, a few more Python command examples are provided in `<VITIS_INSTALL_DIR>/2024.1/cli/examples`.

Note: Accelerated flows are not supported in embedded installer.

For more details of all Vitis supported Python APIs, refer to the link:

`<vitis_install_path>cli/api_docs/build/html/vitis.html`

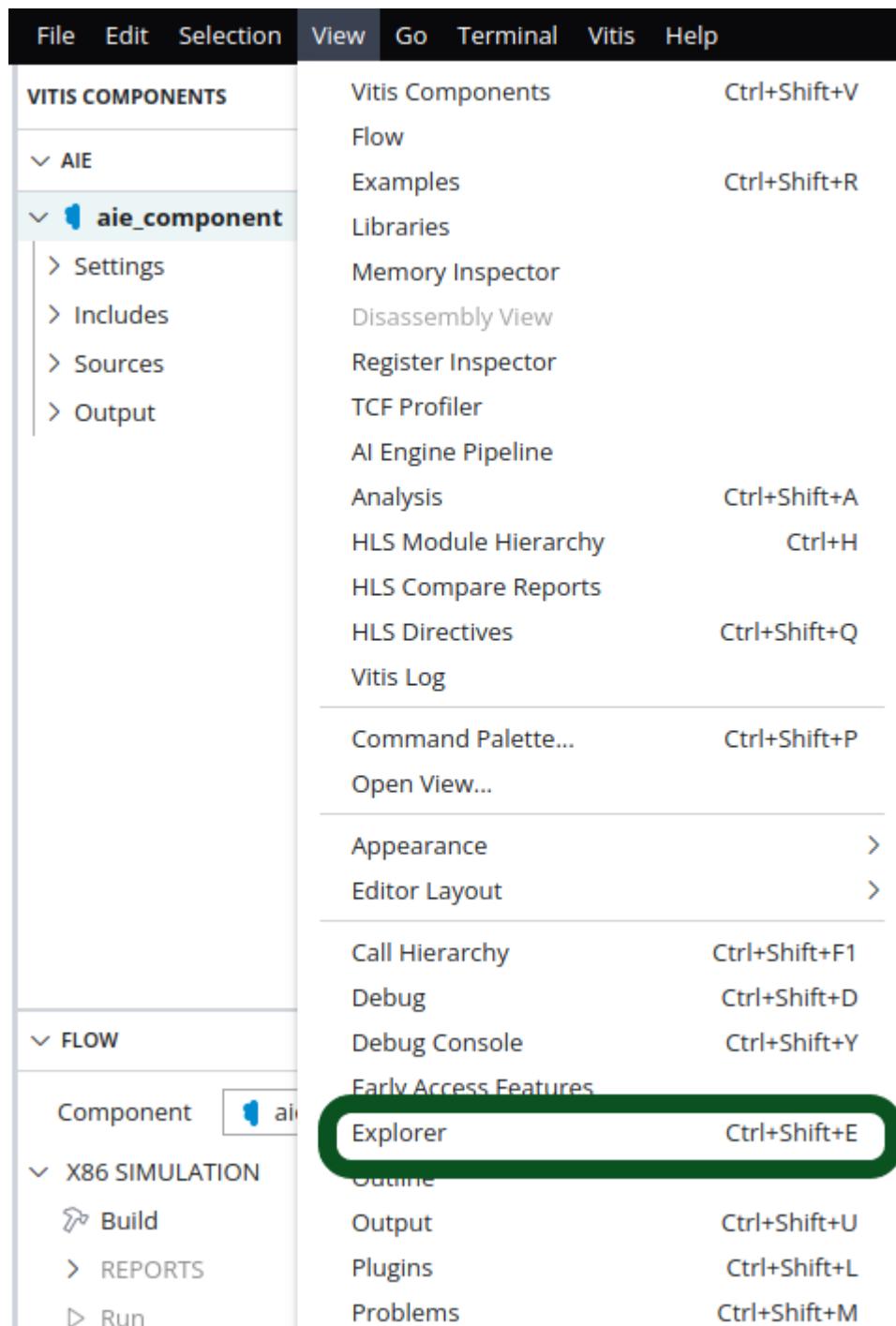
Script Building Logger: Tool to Automate Script Creation Based on IDE Actions

The script building logger converts the Vitis tool GUI user actions (create and build Vitis IDE components such as AI Engine component) into equivalent Python APIs. The Python file generated by the script building logger can be used to create and build components using the Python interpreter.

The script building logger logs actions from the GUI and saves the file as `builder.py`. To access `builder.py`, follow the steps below:

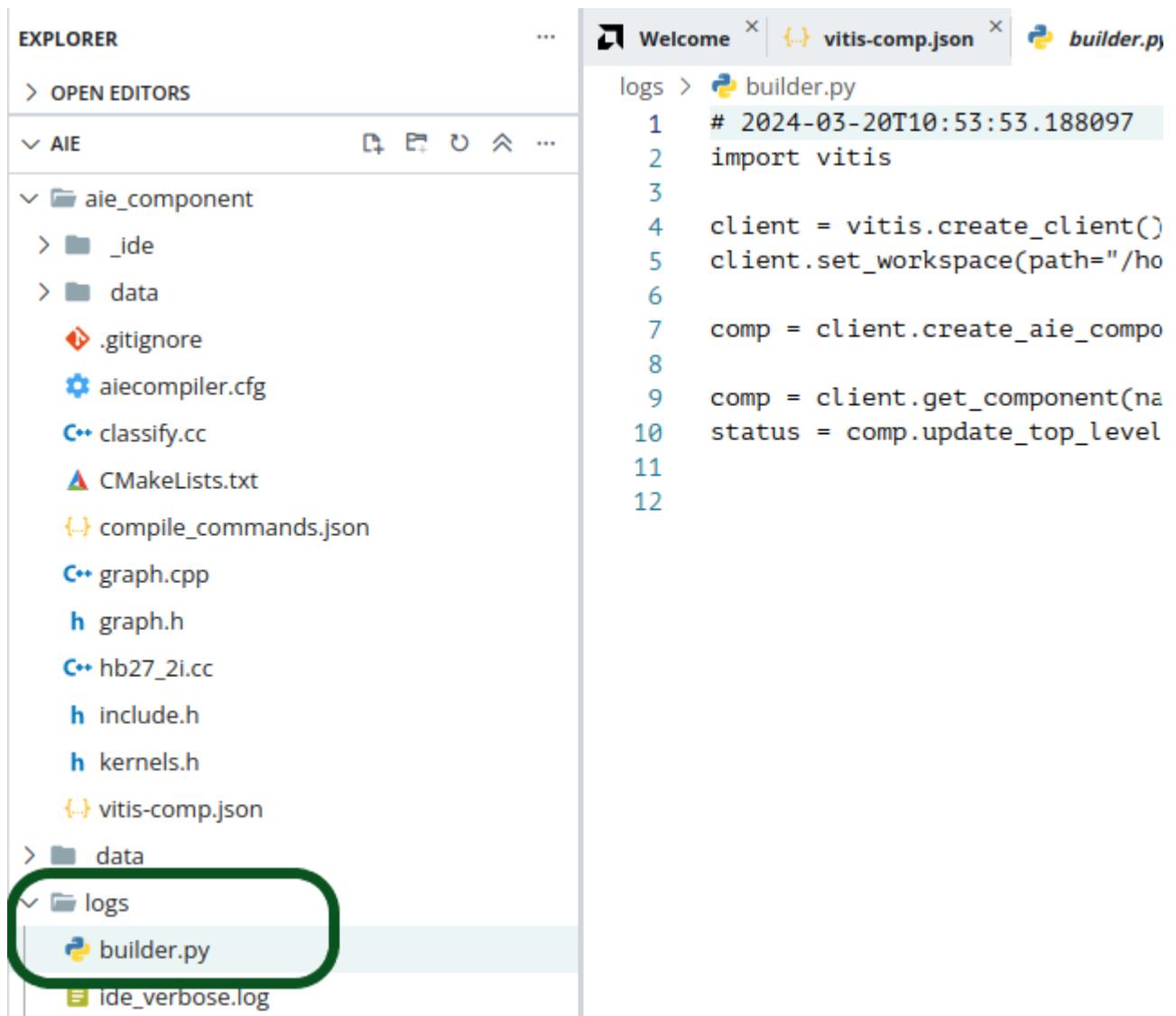
1. Navigate to **View → Explorer** in the Menu bar or press **Ctrl+Shift+E**.

Figure 66: View → Explorer menu



2. In the **Explorer** window, go to `logs: builder.py`. The Python interpreter creates `builder.py`, which logs the equivalent Python APIs for the commands used in the Vitis IDE. You can run `builder.py` later to automate the process.

Figure 67: logs → builder.py



The image shows a code editor interface with the following details:

- EXPLORER** (Left): Shows the project structure:
 - AIE
 - aie_component
 - _ide
 - data
 - .gitignore
 - aiecompiler.cfg
 - classify.cc
 - CMakeLists.txt
 - compile_commands.json
 - graph.cpp
 - graph.h
 - hb27_2i.cc
 - include.h
 - ernels.h
 - vitis-comp.json
 - data
 - logs (highlighted with a green oval)
 - ide_verbose.log
 - Editor Tab** (Right): The 'logs' tab is active, showing the 'builder.py' file content. The code is as follows:

```
# 2024-03-20T10:53:53.188097
import vitis

client = vitis.create_client()
client.set_workspace(path="/ho")
comp = client.create_aie_compo
comp = client.get_component(name="hb27_2i")
status = comp.update_top_level
```

Python Vitis Commands

You can see a list of all the comprehensive Python AMD Vitis™ Commands.

The Vitis CLI can be launched using the following command.

```
vitis -i  
  
client = vitis.create_client()  
help(client)
```

Vitis Python command examples are provided in <VITIS_INSTALL_DIR>/2024.1/cli/examples.

All Vitis Python commands to rebuild a Vitis Workspace are provided in <vitis_workspace>/logs/builder.py.

Python XSDB Commands

You can view a list of all the comprehensive Python XSDB API using the following commands in AMD Vitis™ CLI.

Launch the Vitis CLI using the following command.

```
vitis -i

from xsdb import *
session = start_debug_session()
help("functions")
```

Python XSDB Usage Examples

Debug Operations on Session Object

In this mode, you can create a session object and run debug operations on different debug targets using the same session object.

```
session=start_debug_session()
session.connect(url="TCP:xhdbfarmrkd11:3121")
session.targets(3)
# All subsequent commands are run on target 3, until the target is changed
with targets() function
session.dow("test.elf")
session.bpadd(addr='main')
session.con()
session.targets(4)
# All subsequent commands are run on target 4
session.dow("foo.elf")
session.bpadd(addr='foo')
session.con()
```

Debug Operations on Target Object

You can also use the Target object returned by targets() functions and run debug operations can be performed on these objects. The following is an example:

```
session = start_debug_session()
session.connect(url="TCP:xhdbfarmrkd11:3121")
ta3 = session.targets(3)
ta4 = session.targets(4)
# Run debug commands using target objects
ta3.dow("test.elf")
ta4.dow("foo.elf")
bp1 = ta3.bpadd(addr='main')
bp2 = ta4.bpadd(addr='foo')
ta3.con()
ta4.con()
...
bp1.status()
bp2.status()
```

For interactive usage, it is recommended to use commands and options instead of functions and arguments, as functions require a lot of extra typing. You have defined an `interactive()` function in `xsdb` module, which supports the commands. The following is an example:

```
Vitis-ng [1]: import xsdb
Vitis-ng [2]: xsdb.interactive()
% conn -host xhdbfarmrkb9
tcfchan#0
% ta
1 APU
    2 ARM Cortex-A9 MPCore #0 (Running)
    3 ARM Cortex-A9 MPCore #1 (Running)
4 xc7z020
% ta 2
<xsdpy._target.Target object at 0x7fb2652d3520>
% stop
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffffff28 (Suspended)
% q
Vitis-ng [3]:
```

Programming U-Boot over JTAG

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrkg7:3121")
# Disable Security gates to view PMU MB target
s.targets("--set", filter="name =~ PSU")
# By default, JTAG security gates are enabled
# This disables security gates for DAP, PLTAP and PMU.
s.mwr(0xfffc0038, words=0x1ff)
time.sleep(0.5)
# Load and run PMU FW
s.targets("--set", filter="name =~ MicroBlaze PMU")
s.dow('pmufw.elf')
s.con()
time.sleep(0.5)
# Reset A53, load and run FSBL
s.targets("--set", filter="name =~ Cortex-A53 #0")
s.targets()
s.rst(type='processor')
s.dow('fsbl.elf')
s.con()
# Give FSBL time to run
time.sleep(5)
s.stop()
# Downloading the other Softwares like u-boot..etc using below command
s.dow('system.dtb', '-d', addr=0x100000)
s.dow('u-boot.elf')
s.dow('bl31.elf')
s.con()
time.sleep(5)
s.stop()
```

Generate SVF files

```
# Reset values of respective cores
core = 0
apu_reset_a53 = [0x380e, 0x340d, 0x2c0b, 0x1c07]
# Generate SVF file for linking DAP to the JTAG chain
# Next 2 steps are required only for Rev2.0 silicon and above.
s = xsdb.start_debug_session()
s.connect(url="TCP:xhxxxx41x:3121")
svf = s.svf()
svf.config('--linkdap', scan_chain=[0x14738093, 12, 0x5ba00477, 4],
           device_index=1, out="dapcon.svf")
svf.generate()

svf = s.svf()
# Configure the SVF generation
svf.config(scan_chain=[0x14738093, 12, 0x5ba00477, 4],
           device_index=1, cpu_index=core, delay=10, out="fsbl_hello.svf")
# Record writing of bootloop and release of A53 core from reset
svf.mwr(0xfffff0000, 0x14000000)
svf.mwr(0xfd1a0104, apu_reset_a53[core])
# Record stopping the core
svf.stop()
# Record downloading FSBL
svf.dow(file='zynq_fsbl.elf')
# Record executing FSBL
svf.con()
svf.delay(100000)
# Record some delay and then stopping the core
svf.stop()
# Record downloading the application
svf.dow(file='zynq_hello.elf')
# Record executing application
svf.con()
# Generate SVF
svf.generate()
```

Using readjtaguart function

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrke9:3121")
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
s.targets("--set", filter="name =~ ARM Cortex-A9 MPCore #0")
s.targets()
s.rst()
s.loadhw(hw='zc702.xsa')
# run FSBL for ps7_init
s.dow('fsbl.elf')
s.con()
time.sleep(0.5)
s.stop()
# Download the application program
s.dow('test_jtag_uart.elf')
s.readjtaguart()
s.con()
s.readjtaguart('--stop') # after you are done
```

Using readjtaguart_file function

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrke9:3121")
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
s.targets("--set", filter="name =~ ARM Cortex-A9 MPCore #0")
s.targets()
s.rst()
s.loadhw(hw='zc702.xsa')
# run FSBL for ps7_init
s.dow('fsbl.elf')
s.con()
time.sleep(0.5)
s.stop()
# Download the application program
s.dow('test_jtag_uart.elf')
s.readjtaguart(file='streams.log', mode='w')
s.con()
s.readjtaguart('--stop') # after you are done
```

Using jtag_terminal function

```
s = xsdb.start_debug_session()
# connecting to the target
s.connect(url="TCP:xhdbfarmrke9:3121")
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
s.targets("--set", filter="name =~ ARM Cortex-A9 MPCore #0")
s.targets()
s.rst()
s.loadhw(hw='zc702.xsa')
# run FSBL for ps7_init
s.dow('fsbl.elf')
s.con()
time.sleep(0.5)
s.stop()
# Download the application program
s.dow('test_jtag_uart.elf')
s.jtagterminal()
s.con()
s.jtagterminal('--stop') # after you are done
```

Performing_standalone_app_debug:

```
Vitis [2]: import xsdb
Vitis [3]: s = xsdb.start_debug_session()
# connecting to the target
Vitis [4]: s.connect(url="TCP:xhdbfarmrke9:3121")
tcfchan#0
Out[4]: 'tcfchan#0'
```

```
# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
# For non-interactive usage, -filter option can be used to select a target,
# instead of selecting the target through its ID
Vitis [5]: s.targets()
1 APU
2 ARM Cortex-A9 MPCore #0 (Running)
3 ARM Cortex-A9 MPCore #1 (Running)
4 xc7z020

Vitis [6]: s.targets("--set", filter="name =~ APU")
Out[6]: <xsdb._target.Target at 0x7f01764c6e20>

Vitis [7]: s.targets()
1* APU
2 ARM Cortex-A9 MPCore #0 (Running)
3 ARM Cortex-A9 MPCore #1 (Running)
4 xc7z020

Vitis [8]: s.rst()

Vitis [9]: s.fpga(file='zc702.bit')
100% 3.86MB 1.66MB/s 00:02ETA

Vitis [10]: s.targets(2)
Out[10]: <xsdb._target.Target at 0x7f014d1f1610>

Vitis [11]: s.loadhw(hw='zc702.xsa')
INFO: [Hsi 55-2053] elapsed time for repository (/proj/xbuilds/
HEAD_daily_latest/install/lin64/Vitis/HEAD/data/embeddedsw) loading 1
seconds

# run FSBL for ps_init
Vitis [12]: s.dow('fsbl.elf')
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffffff28 (Suspended)
Downloading Program -- fsbl.elf
section, .text: 0x00000000 - 0x0000fc8b
section, .handoff: 0x0000fc8c - 0x0000fcd7
section, .init: 0x0000fcda - 0x0000fce3
section, .fini: 0x0000fce4 - 0x0000fceef
section, .rodata: 0x0000fcf0 - 0x00010043
section, .data: 0x00010048 - 0x00013017
section, .mmu_tbl: 0x00014000 - 0x00017fff
section, .init_array: 0x00018000 - 0x00018003
section, .fini_array: 0x00018004 - 0x00018007
section, .rsa_ac: 0x00018008 - 0x0001903f
section, .bss: 0x00019040 - 0x0001b3ff
section, .heap: 0x0001b400 - 0x0001d3ff
section, .stack: 0xfffff0000 - 0xfffffd3ff

Successfully downloaded fsbl.elf
Setting PC to Program Start Address 0x00000000

Vitis [13]: s.con()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Vitis [14]: s.stop()

Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xfcac (Suspended)
```

```
# Download the application program
Vitis [15]: s.dow('test_jtag_uart.elf')
Downloading Program -- test_jtag_uart.elf
  section, .text: 0x00100000 - 0x00101503
  section, .init: 0x00101504 - 0x0010150f
  section, .fini: 0x00101510 - 0x0010151b
  section, .rodata: 0x0010151c - 0x001016cc
  section, .data: 0x001016d0 - 0x00101b3f
  section, .eh_frame: 0x00101b40 - 0x00101b43
  section, .mmu_tbl: 0x00104000 - 0x00107fff
  section, .init_array: 0x00108000 - 0x00108003
  section, .fini_array: 0x00108004 - 0x00108007
  section, .bss: 0x00108008 - 0x0010802f
  section, .heap: 0x00108030 - 0x0010a02f
  section, .stack: 0x0010a030 - 0x0010d82f

Successfully downloaded test_jtag_uart.elf
Setting PC to Program Start Address 0x00100000

Vitis [16]: s.bpadd(addr='main')
Out[16]: <xsdb._breakpoint.Breakpoint at 0x7f016fe59af0>

Info: Breakpoint 0 status:
  target 2: {Address: 0x100564 Type: Hardware}
Info: Breakpoint 0 status:
  target 3: {At col 4: Undefined identifier main. Invalid expression}
Vitis [17]: s.con()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100570 (Breakpoint)
main() at ../../src/helloworld.c: 57
57: int l_int_b = 20;
Vitis [18]: s.stp()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100578 (Step)
main() at ../../src/helloworld.c: 58
58: int l_int_a = 40;
Vitis [19]: s.stp()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100580 (Step)
main() at ../../src/helloworld.c: 60
60: init_platform();
Vitis [20]: s.rrd()
      r0: 00000000          r1: 00000000          r2:
00000001          r3: 00000028          r4: 00000003          r5:
0000001e          r6: 0000ffff          r7: f8f00000          r8:
0000767b          r9: ffffffff          r10: 00000000          r11:
0010c02c          r12: 00000000          sp: 0010c020          lr:
001007bc          pc: 00100580          cpsr: 6000005f          usr
          fiq          irq          abt
          und          svc          mon
          vfp          cp15          Jazelle
          gpv_qos301_cpu          gpv_qos301_dmac          gpv_qos301_iou
          gpv_trustzone          l2cache          mpcore
```

```
Vitis [21]: s.mrd(0xe000d000)
E000D000 : 800A0000

# Local variable value can be modified
Vitis [22]: s.locals()
l_int_b : 20
l_int_a : 40

Vitis [23]: s.locals(name='l_int_b', val=815)

Vitis [24]: s.locals(name='l_int_b')
l_int_b : 815

# Global variables and be displayed, and its value can be modified
Vitis [25]: s.print(expr='g_int_a')
g_int_a : 60

Vitis [26]: s.print(expr='g_int_a', val=9919)

Vitis [27]: s.print(expr='g_int_a')
g_int_a : 9919

# Expressions can be evaluated and its value can be displayed
Vitis [28]: s.print('--add', expr='l_int_a + l_int_b')
l_int_a + l_int_b : 855

# Step over a line of source code
Vitis [29]: s.nxt()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x100584 (Step)
main() at ../../src/helloworld.c: 61
61: test_function(l_int_a, l_int_b);

# View stack trace
Vitis [30]: s.bt()
0 0x100584 main() + 32: ../../src/helloworld.c, line 61
1 0x1007bc __start() + 88: xil-crt0.S, line 119
2 unknown-pc

# Set a breakpoint at exit and resume execution
Vitis [31]: s.bpadd(addr='&exit')
Out[29]: <xsdb._breakpoint.Breakpoint at 0x7f016fd2c5e0>

Info: Breakpoint 1 status:
    target 2: {Address: 0x1012ac Type: Hardware}
Info: Breakpoint 1 status:
    target 3: {At col 5: Undefined identifier exit. Invalid expression}
Vitis [32]: s.con()

Info: ARM Cortex-A9 MPCore #0 (target 2) Running
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x1012ac (Breakpoint)

# View stack trace
Vitis [33]: s.bt()
0 0x1012ac exit() + 0
1 0x71034f12
```

Select_target_based_on_target_properties:

```
Vitis [2]: import xsdb

Vitis [3]: s = xsdb.start_debug_session()

# connecting to the target
Vitis [4]: s.connect(url="TCP:xhdbfarmrke9:3121")
tcfchan#0
Out[4]: 'tcfchan#0'

# check the jtag targets connected, the IDs listed with jtag targets are
# called node IDs
Vitis [5]: s.jtag_targets()
1 Digilent JTAG-SMT1 210203344713A ( is_pdi_programmable)
2 arm_dap (idcode 4ba00477 irlen 4 is_pdi_programmable)
3 xc7z020 (idcode 23727093 irlen 6 fpga is_pdi_programmable)

# check the targets connected, the IDs listed with targets are called
# target IDs
Vitis [6]: s.targets()
1 APU
2 ARM Cortex-A9 MPCore #0 (Breakpoint)
3 ARM Cortex-A9 MPCore #1 (Running)
4 xc7z020

# check jtag target properties of 2nd device (2nd ARM DAP). Note the
# target_ctx here.
Vitis [7]: s.jtag_targets('-t', filter="node_id == 2")
Out[7]:
{'jsn-JTAG-SMT1-210203344713A-4ba00477-0': {'target_ctx': 'jsn-JTAG-
SMT1-210203344713A-4ba00477-0',
 'level': 1,
 'node_id': 2,
 'is_open': 1,
 'is_active': 1,
 'is_current': 0,
 'name': 'arm_dap',
 'jtag_cable_name': 'Digilent JTAG-SMT1 210203344713A',
 'state': '',
 'jtag_cable_manufacturer': 'Digilent',
 'jtag_cable_product': 'JTAG-SMT1',
 'jtag_cable_serial': '210203344713A',
 'idcode': '4ba00477',
 'irlen': '4',
 'is_fpga': 0,
 'is_pdi_programmable': 0}

# using the target context, select the targets associated with the JTAG
# target (2nd ARM DAP - node id = 2)
Vitis [8]: s.targets(filter="jtag_device_ctx == jsn-JTAG-
SMT1-210203344713A-4ba0
    ...: 0477-0")
1 APU
2 ARM Cortex-A9 MPCore #0 (Breakpoint)
3 ARM Cortex-A9 MPCore #1 (Running)
```

Debugging_prog_already_running_on_target:

```
Vitis [1]: import xsdb
Vitis [2]: s = xsdb.start_debug_session()

# connecting to the target
Vitis [3]: s.connect(url="TCP:xhdbfarmrke9:3121")
tcfchan#0
Out[3]: 'tcfchan#0'

# Select the target on which the program is running and specify the symbol
# file using the
# memmap command

Vitis [4]: s.target(2)
Out[4]: <xsdb._target.Target at 0x7f3167dd12e0>

Vitis [5]: s.memmap(file='test_jtag_uart.elf')

# When the symbol file is specified, the debugger maps the code on the
target to the symbol
# file. bt command can be used to see the back trace. Further debug is
possible, as shown in
# the first example

Vitis [6]: s.stop()
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0x10101c (Suspended)
sleep_A9() at sleep.c: 63
63: } while (tCur < tEnd);

Vitis [7]: s.backtrace()
0 0x10101c sleep_A9() +56:sleep.c, line 63
1 0x100608 test_function() +76:../src/helloworld.c, line 78
2 0x100590 main() +44:../src/helloworld.c, line 61
3 0x1007ac __start() +88:xil-crt0.S, line 119
4 unknown -pc
```

Debug_app_on_zu_plus_mpsoc:

```
Vitis [1]: import xsdb
Vitis [2]: s = xsdb.start_debug_session()

# connect to remote hw_server by specifying its url.
# If the hardware is connected to a local machine, -url option and the <url>
# are not needed. connect command returns the channel ID of the connection
Vitis [3]: s.connect(url="TCP:xhdbfarmrkk5:3121")
tcfchan#0
Out[3]: 'tcfchan#0'

# List available targets and select a target through its id.
# The targets are assigned IDs as they are discovered on the Jtag chain,
# so the IDs can change from session to session.
Vitis [4]: s.targets()
1 PS TAP
2 PMU
3 PL
4 PSU
5 RPU (Reset)
6 Cortex-R5 #0 (RPU Reset)
```

```
    7  Cortex-R5 #1 (RPU Reset)
  8  APU (L2 Cache Reset)
    9  Cortex-A53 #0 (APU Reset)
  10  Cortex-A53 #1 (APU Reset)
  11  Cortex-A53 #2 (APU Reset)
  12  Cortex-A53 #3 (APU Reset)

Vitis [5]: s.targets("--set", filter="name =~ PSU")
Out[5]: <xsdb._target.Target at 0x7fe9ca93bac0>

Vitis [6]: s.taa()
  1  PS TAP
  2  PMU
  3  PL
  4* PSU
    5  RPU (Reset)
      6  Cortex-R5 #0 (RPU Reset)
      7  Cortex-R5 #1 (RPU Reset)
  8  APU (L2 Cache Reset)
    9  Cortex-A53 #0 (APU Reset)
  10  Cortex-A53 #1 (APU Reset)
  11  Cortex-A53 #2 (APU Reset)
  12  Cortex-A53 #3 (APU Reset)

Vitis [7]: s.rst(type='system')

# Configure the FPGA. When the active target is not a FPGA device,
# the first FPGA device is configured
Vitis [8]: s.fpga(file='zcu102.bit')
100% 3.86MB 1.66MB/s 00:02ETA

Vitis [9]: s.targets(10)
Out[9]: <xsdb._target.Target at 0x7fe9a99347f0>

Vitis [10]: s.rst(type='cores')

Info: Cortex-A53 #0 (target 9) Stopped at 0xfffff0000 (Reset Catch)
Info: Cortex-A53 #1 (target 10) Stopped at 0xfffff0000 (Reset Catch)
Info: Cortex-A53 #2 (target 11) Stopped at 0xfffff0000 (Reset Catch)
Info: Cortex-A53 #3 (target 12) Stopped at 0xfffff0000 (Reset Catch)

# run fsbl to initialize PS
Vitis [11]: s.dow('fsbl_a53.elf')
Downloading Program -- fsbl_a53.elf
  section, .text: 0xfffffc0000 - 0xffffd65b7
  section, .note.gnu.build-id: 0xffffd65b8 - 0xffffd65db
  section, .init: 0xfffffd6600 - 0xffffd6633
  section, .fini: 0xfffffd6640 - 0xffffd6673
  section, .rodata: 0xfffffd6680 - 0xffffd6b94
  section, .sys_cfg_data: 0xfffffd6bc0 - 0xfffffd732b
  section, .mmu_tb10: 0xfffffd8000 - 0xffffd800f
  section, .mmu_tb11: 0xfffffd9000 - 0xffffdafff
  section, .mmu_tb12: 0xfffffdb000 - 0xffffdefff
  section, .data: 0xfffffd1000 - 0xffffe02c7
  section, .sbss: 0xffffe02c8 - 0xffffe02ff
  section, .bss: 0xffffe0300 - 0xffffe297f
  section, .heap: 0xffffe2980 - 0xffffe2d7f
  section, .stack: 0xffffe2d80 - 0xffffe4d7f
  section, .dup_data: 0xffffe4d80 - 0xffffe6047
  section, .handoff_params: 0xffffe9e00 - 0xffffe9e87
  section, .bitstream_buffer: 0xfffff0040 - 0xfffffc3f
```

```
Successfully downloaded fsbl_a53.elf
Setting PC to Program Start Address 0xffffc0000

Vitis [12]: s.con()

Info: Cortex-A53 #1 (target 10) Running
Vitis [13]: s.stop()

Info: Cortex-A53 #1 (target 10) Stopped at 0xffffd2c7c (External Debug
Request)

# run the application
Vitis [14]: s.dow('zcu102_hello.elf')
Downloading Program -- zcu102_hello.elf
  section, .text: 0x00000000 - 0x000020b3
  section, .init: 0x000020c0 - 0x000020f3
  section, .fini: 0x00002100 - 0x00002133
  section, .rodata: 0x00002140 - 0x00002360
  section, .rodata1: 0x00002361 - 0x0000237f
  section, .sdata2: 0x00002380 - 0x0000237f
  section, .sbss2: 0x00002380 - 0x0000237f
  section, .data: 0x00002380 - 0x00002bb7
  section, .data1: 0x00002bb8 - 0x00002bbf
  section, .note.gnu.build-id: 0x00002bc0 - 0x00002be3
  section, .ctors: 0x00002be4 - 0x00002bff
  section, .dtors: 0x00002c00 - 0x00002bff
  section, .eh_frame: 0x00002c00 - 0x00002c03
  section, .mmu_tb10: 0x00003000 - 0x0000300f
  section, .mmu_tb11: 0x00004000 - 0x00005fff
  section, .mmu_tb12: 0x00006000 - 0x00009fff
  section, .preinit_array: 0x0000a000 - 0x00009fff
  section, .init_array: 0x0000a000 - 0x0000a007
  section, .fini_array: 0x0000a008 - 0x0000a047
  section, .sdata: 0x0000a048 - 0x0000a07f
  section, .sbss: 0x0000a080 - 0x0000a07f
  section, .tdata: 0x0000a080 - 0x0000a07f
  section, .tbss: 0x0000a080 - 0x0000a07f
  section, .bss: 0x0000a080 - 0x0000a0bf
  section, .heap: 0x0000a0c0 - 0x0000c0bf
  section, .stack: 0x0000c0c0 - 0x0000f0bf

Successfully downloaded zcu102_hello.elf
Setting PC to Program Start Address 0x00000000

Vitis [15]: s.bpadd(addr='main')
Out[15]: <xsdb._breakpoint.Breakpoint at 0x7fe9cae59cd0>

Info: Breakpoint 0 status:
  target 6: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
  target 7: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
  target 9: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
  target 10: {Address: 0xd00 Type: Hardware}
Info: Breakpoint 0 status:
  target 11: {At col 4: Undefined identifier main. Invalid expression}
Info: Breakpoint 0 status:
  target 12: {At col 4: Undefined identifier main. Invalid expression}
Vitis [16]: s.con()

Info: Cortex-A53 #1 (target 10) Running
Info: Cortex-A53 #1 (target 10) Stopped at 0xd08 (Breakpoint)
```

```
main() at ../../src/helloworld.c: 29
29: int l_int_b = 20;
Vitis [17]: s.rrd()
    r0: 00000000          r1: 00000000          r2: 00000000          r3:
00000004
    r4: 0000000f          r5: ffffffff          r6: 0000001c          r7:
00000002
    r8: ffffffff          r9: 00000000          r10: fd5c0090         r11:
00000000
    r12: 00000000         r13: 00000000         r14: 00000000         r15:
00000000
    r16: 00000000         r17: 00000000         r18: 00000000         r19:
00000000
    r20: 00000000         r21: 00000000         r22: 00000000         r23:
00000000
    r24: 00000000         r25: 00000000         r26: 00000000         r27:
00000000
    r28: 00000000         r29: 0000e0a0         r30: 00000f34         sp:
0000e0a0
    pc: 00000d08          cpsr: 600002cd          vfp
    dbg          acpu_gic          sys

Vitis [18]: s.stp()

Info: Cortex-A53 #1 (target 10) Running
Info: Cortex-A53 #1 (target 10) Stopped at 0xd10 (Step)
main() at ../../src/helloworld.c: 30
30: int l_int_a = 40;
Vitis [19]: s.stp()

Info: Cortex-A53 #1 (target 10) Running
Info: Cortex-A53 #1 (target 10) Stopped at 0xd18 (Step)
main() at ../../src/helloworld.c: 32
32: init_platform();

# Local variable value can be modified
Vitis [20]: s.locals()
l_int_b : 20
l_int_a : 40

Vitis [21]: s.locals(name='l_int_b', val=815)

Vitis [22]: s.locals(name='l_int_b')
l_int_b : 815

# Global variables and be displayed, and its value can be modified
Vitis [23]: s.print(expr='g_int_a')
g_int_a : 60

Vitis [24]: s.print(expr='g_int_a', val=9919)

Vitis [25]: s.print(expr='g_int_a')
g_int_a : 9919

# Expressions can be evaluated and its value can be displayed
Vitis [26]: s.print('--add', expr='l_int_a + l_int_b')
l_int_a + l_int_b : 855

# Step over a line of source code
Vitis [27]: s.nxt()

Info: Cortex-A53 #1 (target 10) Running
```

```
Info: Cortex-A53 #1 (target 10) Stopped at 0xd1c (Step)
main() at ../../src/helloworld.c: 33
33: test_function(l_int_a, l_int_b);

# View stack trace
Vitis [28]: s.bt()
0 0xd1c main() +28:../../src/helloworld.c, line 33
1 0xf34 _startup() +124:xil-crt0.S, line 157

# Set a breakpoint at exit and resume execution
Vitis [29]: s.bpadd(addr='&exit')
Info: Breakpoint 1 status:
    target 6: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
    target 7: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
    target 9: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
    target 10: {Address: 0x1a90 Type: Hardware}
Info: Breakpoint 1 status:
    target 11: {At col 5: Undefined identifier exit. Invalid expression}
Info: Breakpoint 1 status:
    target 12: {At col 5: Undefined identifier exit. Invalid expression}
Out[29]: <xsdb._breakpoint.Breakpoint at 0x7fe9a35e9ca0>

Vitis [30]: s.con()

Info: Cortex-A53 #1 (target 10) Running

Vitis [31]: s.stop()
Info: Cortex-A53 #1 (target 10) Stopped at 0x192c (External Debug Request)
sleep_A53() at sleep.c: 71
71: } while (tCur < tEnd);

# View stack trace
Vitis [32]: s.bt()
0 0x192c sleep_A53() +44:sleep.c, line 71
1 0x192c sleep_A53() +44:sleep.c, line 70
```

Software Command-Line Tool

This section contains the following chapters:

- [Software Command-Line Tool](#)
- [XSCT Commands](#)
- [XSCT Use Cases](#)
- [Hardware Software Interface \(HSI\) Commands](#)

Software Command-Line Tool

The AMD Vitis™ IDE is a graphical development environment that is helpful when developing for a new processor architecture. It simplifies common functions through logical wizards, so even beginners can use it. However, it is necessary for these tools to be scriptable, meaning they can be modified or extended, providing flexibility. This is especially helpful for developing regression tests that are run daily or when using a specific set of commands frequently. It is particularly useful when developing regression tests that are run nightly or when running a set of commands that are frequently used.

Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to the Vitis IDE. As with other AMD tools, the scripting language for XSCT is based on the tools command language (Tcl). You can run XSCT commands interactively or script the commands for automation.

XSCT supports Vitis project management, configuration, building, and debugging, such as:

- Creating platform projects and domains
- Creating system and application projects
- Configuring and building domains/BSPs and applications
- Managing repositories
- Setting toolchain preferences
- Downloading and running applications on hardware targets
- Reading and writing registers
- Setting break points and watch expressions

This reference guide is intended to provide the information you need to develop scripts for software development and debug targeting AMD processors.

In this guide, abbreviations are used for various products produced by AMD. For example:

- Use of `ps7` in the source code implies that these files are targeting the AMD Zynq™ 7000 SoC family of products, and specifically the dual-core Cortex® Arm® A9 processors in the SoC.
- Use of `psu` in the source code implies that this code is targeting an AMD Zynq™ UltraScale+™ MPSoC device, which contains a Cortex quad-core Arm A53, dual-core Arm R5, Arm Mali 400 GPU, and a MicroBlaze™ processor based platform management unit (PMU).

- Hardware definition files (XSA) are used to transfer the information about the hardware system that includes a processor to the embedded software development tools such as Vitis IDE and Software Command-Line Tools (XSCT). It includes information about which peripherals are instantiated, clocks, memory interfaces, and memory maps.
- Microprocessor Software Specification (MSS) files are used to store information about the domain/BSP. They contain OS information for the domain/BSP, software drivers associated with each peripheral of the hardware design, STDIO settings, and compiler flags such as optimization and debug information level.

XSCT Commands

The XSCT commands are described in the following sections.

Target Connection Management

The following is a list of connections commands:

- [connect](#)
- [disconnect](#)
- [targets](#)
- [gdbremote connect](#)
- [gdbremote disconnect](#)

connect

Connect to hw_server/TCF agent.

Syntax

```
connect [options]
```

Allows users to connect to a server, list connections, or switch between connections.

Options

Option	Description
<code>-host <host name/ip></code>	Name/IP address of the host machine.
<code>-port <port num></code>	TCP port number.
<code>-url <url></code>	URL description of hw_server/TCF agent.
<code>-list</code>	List open connections.
<code>-set <channel-id></code>	Set active connection.
<code>-new</code>	Create a new connection, even one existing to the same URL.
<code>-xvc-url <url></code>	Open Xilinx virtual cable connection.

Option	Description
-symbols	Launch symbol server to enable source-level debugging for remote connections.

Returns

The return value depends on the options used.

-port, -host, -url, -new:<channel-id> of the new connection or error if the connection fails.

-list: List of open channels or nothing when there are no open channels.

-set: Nothing.

Examples

```
connect -host localhost -port 3121
```

Connect to hw_server/TCF agent on host localhost and port 3121.

```
connect -url tcp:localhost:3121
```

Identical to the previous example.

disconnect

Disconnect from hw_server/TCF agent.

Syntax

```
disconnect
```

Disconnect from active channel.

```
disconnect <channel-id>
```

Disconnect from specified channel.

Returns

Nothing, if the connection is closed. Error string, if invalid channel-id is specified.

targets

List targets or switch between targets.

Syntax

```
targets [options]
```

List available targets.

```
targets <target id>
```

Select <target id> as active target.

Options

Option	Description
-set	Set current target to entry single entry in list. This is useful in combination with the -filter option. An error is generated if the list is empty or contains more than one entry.
-regexp	Use regexp for filter matching
-nocase	Use case insensitive filter matching
-filter <filter-expression>	Specify filter expression to control which targets are included in list based on its properties. Filter expressions are similar to Tcl expr syntax. Target properties are referenced by name, while Tcl variables are accessed using the \$ syntax string must be quoted. Operators ==, !=, <=, >=, <, >, &&, , () are supported. These operators behave like Tcl expr operators. String matching operators =~ and !~ match the LHS string with the RHS pattern using either regexp or string match.
-target-properties	Returns a Tcl list of dicts containing target properties.
-index <index>	Include targets based on the JTAG scan chain position. This is identical to specifying -filter {jtag_device_index==<index>}.
-timeout <sec>	Poll until the targets specified by filter option are found on the scan chain, or until timeout. This option is valid only with filter option. This option is useful in case of soft processors on PL, as their initialization and detection takes some time. The timeout value is in seconds. Default timeout is three seconds.

Returns

The return value depends on the options used.

<none>: Targets list when no options are used.

-filter: Filtered targets list.

-target-properties: Tcl list consisting of target properties.

An error is returned when target selection fails.

Examples

```
targets
```

List all targets.

```
targets -filter {name =~ "ARM*#1"}
```

List targets with name starting with "ARM" and ending with "#1".

```
targets 2
```

Set target with id 2 as the current target.

```
targets -set -filter {name =~ "ARM*#1"}
```

Set current target to target with name starting with "ARM" and ending with "#1".

```
targets -set -filter {name =~ "MicroBlaze*"} -index 0
```

Set current target to target with name starting with "MicroBlaze" and which is on the first JTAG device.

gdbremote connect

Connect to GDB remote server.

Syntax

```
gdbremote connect [options] server
```

Connect to a GDB remote server (for example, qemu). xrt_server is used to connect to the remote GDB server.

Options

Option	Description
<code>-architecture <name></code>	Specify default architecture if remote server does not provide it.

Returns

Nothing, if the connection is successful. Error string, if the connection fails.

gdbremote disconnect

Disconnect from GDB remote server.

Syntax

```
gdbremote disconnect [target-id]
```

Disconnect from GDB remote server (for example, qemu).

Returns

Nothing, if the connection is close. Error string, if there is no active connection.

Target Registers

The following is a list of registers commands:

- [rrd](#)
- [rwr](#)

rrd

Read register for active target.

Syntax

```
rrd [options] [reg]
```

Read registers or register definitions. For a processor core target, the processor core register can be read. For a target representing a group of processor cores, system registers or IOU registers can be read.

Options

Option	Description
<code>-defs</code>	Read register definitions instead of values.
<code>-no-bits</code>	Does not show bit fields along with register values. By default, bit fields are shown, when available.

Returns

Register names and values, or register definitions if successful. Error string, if the registers cannot be read or if an invalid register is specified.

Examples

```
rrd
```

Read top level registers or groups.

```
rrd r0
```

Read register r0.

```
rrd usr r8
```

Read register r8 in group usr.

rwr

Write to register.

Syntax

```
rwr <reg> <value>
```

Write the `<value>` to active target register specified by `<reg>`. For a processor core target, the processor core register can be written to. For a target representing a group of processor cores, system registers or IOU registers can be written to.

Returns

Nothing, if successful. Error string, if an invalid register is specified or the register cannot be written to.

Examples

```
rwr r8 0x0
```

Write 0x0 to register r8.

```
rwr usr r8 0x0
```

Write 0x0 to register r8 in group usr.

Program Execution

The following is a list of running commands:

- [state](#)
- [stop](#)
- [con](#)

- `stp`
- `nxt`
- `stpi`
- `nxti`
- `stfout`
- `dis`
- `print`
- `locals`
- `backtrace`
- `bt`
- `profile`
- `mbprofile`
- `mbtrace`

state

Display the current state of the target.

Syntax

```
state
```

Return the current execution state of target.

stop

Stop active target.

Syntax

```
stop
```

Suspend execution of active target.

Returns

Nothing, if the target is suspended. Error string, if the target is already stopped or cannot be stopped.

An information message is printed on the console when the target is suspended.

con

Resume active target.

Syntax

```
con [options]
```

Resume execution of active target.

Options

Option	Description
-addr <address>	Resume execution from address specified by <address>.
-block	Block until the target stops or a timeout is reached.
-timeout <sec>	Timeout value in seconds.

Returns

Nothing, if the target is resumed. Error string, if the target is already running or cannot be resumed or does not halt within timeout after being resumed.

An information message is printed on the console when the target is resumed.

Examples

```
con -addr 0x100000
```

Resume execution of the active target from address 0x100000.

```
con -block
```

Resume execution of the active target and wait until the target stops.

```
con -block -timeout 5
```

Resume execution of the active target and wait until the target stops or until the five second timeout is reached.

stp

Step into a line of source code.

Syntax

```
stp [count]
```

Resume execution of the active target until control reaches instruction that belongs to different line of source code. If a function is called, stop at first line of the function code. Error is returned if line number information not available. If `<count>` is greater than 1, repeat `<count>` times. Default value of count is 1.

Returns

Nothing, if the target has single stepped. Error string, if the target is already running or cannot be resumed.

An information message is printed on the console when the target stops at the next address.

nxt

Step over a line of source code.

Syntax

```
nxt [count]
```

Resume execution of the active target until control reaches instruction that belongs to a different line of source code, but runs any functions called at full speed. Error is returned if line number information not available. If `<count>` is greater than 1, repeat `<count>` times. Default value of count is 1.

Returns

Nothing, if the target has stepped to the next source line. Error string, if the target is already running or cannot be resumed.

An information message is printed on the console when the target stops at the next address.

stpi

Execute a machine instruction.

Syntax

```
stpi [count]
```

Execute a single machine instruction. If the instruction is a function call, stop at the first instruction of the function code. If `<count>` is greater than 1, repeat `<count>` times. Default value of count is 1.

Returns

Nothing, if the target has single stepped. Error if the target is already running or cannot be resumed.

An information message is printed on the console when the target stops at the next address.

nxti

Step over a machine instruction.

Syntax

```
nxti [count]
```

Step over a single machine instruction. If the instruction is a function call, execution continues until control returns from the function. If `<count>` is greater than 1, repeat `<count>` times. Default value of count is 1.

Returns

Nothing, if the target has stepped to the next address. Error string, if the target is already running or cannot be resumed.

An information message is printed on the console when the target stops at the next address.

stfout

Step out from current function.

Syntax

```
stfout [count]
```

Resume execution of current target until control returns from current function. If `<count>` is greater than 1, repeat `<count>` times. Default value of count is 1.

Returns

Nothing, if the target has stepped out of the current function. Error if the target is already running or cannot be resumed.

An information message is printed on the console when the target stops at the next address.

dis

Disassemble instructions.

Syntax

```
dis <address> [num]
```

Disassemble `<num>` instructions at address specified by `<address>`. The keyword "pc" can be used to disassemble instructions at the current PC. Default value for `<num>` is 1.

Returns

Disassembled instructions if successful. Error string, if the target instructions cannot be read.

Examples

```
dis
```

Disassemble an instruction at the current PC value.

```
dis pc 2
```

Disassemble two instructions at the current PC value.

```
dis 0x0 2
```

Disassemble two instructions at address 0x0.

print

Get or set the value of an expression.

Syntax

```
print [options] [expression]
```

Get or set the value of an expression specified by `<expression>`. The `<expression>` can include constants, local/global variables, CPU registers, or any operator, but pre-processor macros defined through `#define` are not supported. CPU registers can be specified in the format `{$r1}`, where `r1` is the register name. Elements of complex data types, like structures, can be accessed through the `".` operator. For example, the `var1.int_type` refers to the `int_type` element in the `var1` struct. Array elements can be accessed through their indices. For example, `array1[0]` refers to the element at index 0 in `array1`.

Options

Option	Description
<code>-add <expression></code>	Add the <code><expression></code> to the auto expression list. The values or definitions of the expressions in the auto expression list are displayed when the expression name is not specified. Frequently used expressions should be added to the auto expression list.
<code>-defs [expression]</code>	Return the expression definitions like address, type, size, and RW flags. Not all definitions are available for all the expressions. For example, the address is available only for variables and not when the expression includes an operator.
<code>-dict [expression]</code>	Return the result in Tcl dict format, with variable names as dict keys and variable values as dict values. For complex data like structures, names are in the form of parent.child.
<code>-remove [expression]</code>	Remove the expression from auto expression list. Only expressions previously added to the list through <code>-add</code> option can be removed. When the expression name is not specified, all the expressions in the auto expression list are removed.
<code>-set <expression></code>	Set the value of a variable. It is not possible to set the value of an expression which includes constants or operators.

Returns

The return value depends on the options used.

`-add` or `<none>`: Expression value(s)

`-defs`: Expression definition(s)

`-remove` or `-set`: Nothing.

Error string, if the expression value cannot be read or set.

Examples

```
print Int_Glob
```

Return the value of variable `Int_Glob`.

```
print -a Microseconds
```

Add the variable `Microseconds` to auto expression list and return its value.

```
print -a Int_Glob*2 + 1
```

Add the expression (`Int_Glob*2 + 1`) to auto expression list and return its value.

```
print tmp_var.var1.int_type
```

Return the value of int_type element in var1 struct, where var1 is a member of tmp_var struct.

```
print tmp_var.var1.array1[0]
```

Return the value of the element at index 0 in array array1. array1 is a member of var1 struct, which is in turn a member of tmp_var struct.

```
print
```

Return the values of all the expressions in auto expression list.

```
print -defs
```

Return the definitions of all the expressions in auto expression list.

```
print -set Int_Glob 23
```

Set the value of the variable Int_Glob to 23.

```
print -remove Microseconds
```

Remove the expression Microseconds from auto expression list.

```
print {$r1}
```

Return the value of CPU register r1.

locals

Get or set the value of a local variable.

Syntax

```
locals [options] [variable-name [variable-value]]
```

Get or set the value of a variable specified by <variable-name>. When the variable name and value are not specified, values of all the local variables are returned. Elements of complex data types like structures can be accessed through the '.' operator. For example, the var1.int_type refers to the int_type element in the var1 struct. Array elements can be accessed through their indices. For example, array1[0] refers to the element at index 0 in array1.

Options

Option	Description
-defs	Return the variable definitions like address, type, size, and RW flags.

Option	Description
-dict	Return the result in Tcl dict format, with variable names as dict keys and variable values as dict values. For complex data like structures, names are in the form of parent.child.

Returns

The return value depends on the options used.

<none>: Variable value(s)

-defs: Variable definition(s)

Nothing, when variable value is set. Error string, if variable value cannot be read or set.

Examples

```
locals Int_Loc
```

Return the value of the local variable Int_Loc.

```
locals
```

Return the values of all the local variables in the current stack frame.

```
locals -defs
```

Return definitions of all the local variables in the current stack frame.

```
locals Int_Loc 23
```

Set the value of the local variable Int_Loc to 23.

```
locals tmp_var.var1.int_type
```

Return the value of the int_type element in the var1 struct, where var1 is a member of the tmp_var struct.

```
locals tmp_var.var1.array1[0]
```

Return the value of the element at index 0 in array array1. array1 is a member of the var1 struct, which is in turn a member of the tmp_var struct.

backtrace

Stack back trace.

Syntax

```
backtrace [options]
```

Return stack trace for current target. Target must be stopped. Use debug information for best result. The alias for backtrace is 'bt' and can be used interchangeably.

Options

Option	Description
<code>-maxframes <num></code>	Maximum number of frames in stack trace. The default value is 10. The actual number of frames could be less depending on program state. To read all the available frames, use -1.

Returns

Stack trace, if successful. Error string, if stack trace cannot be read from the target.

Examples

```
bt
```

Return top 10 frames from stack trace.

```
bt -maxframes 5
```

Return top 5 frames from stack trace.

```
bt -maxframes -1
```

Return all the available frames from stack trace.

bt

Stack back trace.

Syntax

backtrace [options] Return stack trace for current target. Target must be stopped. Use debug information for best result. The alias for backtrace is 'bt' and can be used interchangeably.

Options

Option	Description
<code>-maxframes <num></code>	Maximum number of frames in stack trace. The default value is 10. The actual number of frames could be less depending on program state. To read all the available frames, use -1.

Returns

Stack trace, if successful. Error string, if stack trace cannot be read from the target.

Examples

```
bt
```

Return top 10 frames from stack trace.

```
bt -maxframes 5
```

Return top 5 frames from stack trace.

```
bt -maxframes -1
```

Return all the available frames from stack trace.

profile

Configure and run the GNU profiler.

Syntax

```
profile [options]
```

Configure and run the GNU profiler. Profiling must be enabled while building the BSP and application to be profiled.

Options

Option	Description
<code>-freq <sampling-freq></code>	Sampling frequency.
<code>-scratchaddr <addr></code>	Scratch memory for storing the profiling related data. It needs to be assigned carefully, because it should not overlap with the program sections.
<code>-out <file-name></code>	Name of the output file for writing the profiling data. This option also runs the profiler and collects the data. If a file name is not specified, profiling data is written to gmon.out.

Returns

Depends on options used.

-scratchaddr, **-freq**: Returns nothing on successful configuration. Error string, in case of error.

-out: Returns nothing, and generates a file. Error string, in case of error.

Examples

```
profile -freq 10000 -scratchaddr 0
```

Configure the profiler with a sampling frequency of 10000 and scratch memory at 0x0.

```
profile -out testgmon.out
```

Output the profile data in testgmon.out.

mbprofile

Configure and run the MB profiler.

Syntax

```
mbprofile [options]
```

Configure and run the MB profiler, a non-intrusive profiler for profiling the application running on MicroBlaze. The output file is generated in gmon.out format. The results can be viewed using the gprof editor. In case of cycle count, an annotated disassembly file is also generated clearly marking the time taken for execution of instructions.

Options

Option	Description
-low <addr>	Low address of the profiling address range.
-high <addr>	High address of the profiling address range.
-freq <value>	MicroBlaze clock frequency in Hz. Default is 100 MHz.
-count-instr	Count number of executed instructions. By default, the number of clock cycles of executed instructions are counted.
-cumulate	Cumulative profiling. Profiling without clearing the profiling buffers.
-start	Enable and start profiling.
-stop	Disable/stop profiling.
-out <filename>	Output profiling data to file. <filename> Name of the output file for writing the profiling data. If the file name is not specified, profiling data is written to gmon.out.

Returns

Depends on options used. -low, -high, -freq, -count-instr, -start, -cumulate Returns nothing on successful configuration. Error string, in case of error.

-stop: Returns nothing, and generates a file. Error string, in case of error.

Examples

```
mbprofile -low 0x0 -high 0x3FFF
```

Configure the mb-profiler with address range 0x0 to 0x3FFF for profiling to count the clock cycles of executed instructions.

```
mbprofile -start
```

Enable and start profiling.

```
mbprofile -stop -out testgmon.out
```

Output the profile data in testgmon.out.

```
mbprofile -count-instr
```

Configure the mb-profiler to profile for entire program address range to count the number of instructions executed.

mbtrace

Configure and run MB trace.

Syntax

```
mbtrace [options]
```

Configure and run MB program and event trace for tracing the application running on MB. The output is the disassembly of the executed program.

Options

Option	Description
-start	Enable and start trace. After starting trace the execution of the program is captured for later output.
-stop	Stop and output trace.
-con	Output trace after resuming execution of active target until a breakpoint is hit. At least one breakpoint or watchpoint must be set to use this option. This option is only available with embedded trace.

Option	Description
<code>-stp</code>	Output trace after resuming execution of the active target until control reaches instruction that belongs to different line of source code.
<code>-nxt</code>	Output trace after resuming execution of the active target until control reaches instruction that belongs to a different line of source code, but runs any functions called at full speed.
<code>-out <filename></code>	Output trace data to a file. <code><filename></code> Name of the output file for writing the trace data. If not specified, data is output to standard output.
<code>-level <level></code>	Set the trace level to "full", "flow", "event", or "cycles". If not specified, "flow" is used.
<code>-halt</code>	Set to halt program execution when the trace buffer is full. If not specified, trace is stopped but program execution continues.
<code>-save</code>	Set to enable capture of load and get instruction new data value.
<code>-low <addr></code>	Set low address of the external trace buffer address range. The address range must indicate an unused accessible memory space. Only used with external trace.
<code>-high <addr></code>	Set high address of the external trace buffer address range. The address range must indicate an unused accessible memory space. Only used with external trace.
<code>-format <format></code>	Set external trace data format to "mdm", "ftm", or "tpiu". If format is not specified, "mdm" is used. The "ftm" and "tpiu" formats are output by Zynq 7000 PS. Only used with external trace.

Returns

Depends on options used. `-start`, `-out`, `-level`, `-halt` `-save`, `-low`, `-high`, `-format` Returns nothing on successful configuration. Error string, in case of error.

`-stop`, `-con`, `-stp`, `-nxt`: Returns nothing, and outputs trace data to a file or standard output. Error string, in case of error.

Examples

```
mbtrace -start
```

Enable and start trace.

```
mbtrace -start -level full -halt
```

Enable and start trace, configuring to save complete trace instead of only program flow and to halt execution when trace buffer is full.

```
mbtrace -stop
```

Stop trace and output data to standard output.

```
mbtrace -stop -out trace.out
```

Stop trace and output data to trace.out.

```
mbtrace -con -out trace.out
```

Continue execution and output data to trace.out.

Target Memory

The following is a list of memory commands:

- [mrd](#)
- [mwr](#)
- [osa](#)
- [memmap](#)

mrd

Memory read.

Syntax

```
mrd [options] <address> [num]
```

Read `<num>` data values from the active target's memory address specified by `<address>`.

Options

Option	Description
<code>-force</code>	Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.

Option	Description
-size <access-size>	<access-size> can be one of the values below: b = Bytes accesses h = Half-word accesses w = Word accesses d = Double-word accesses Default access size is w. Address is aligned to access-size before reading memory, if the '-unaligned-access' option is not used. For targets that do not support double-word access, the debugger uses two word accesses. If the number of data values to be read is more than 1, the debugger selects the appropriate access size. For example, 1. mrd -size b 0x0 4 Debugger accesses one word from the memory, displays four bytes. 2. mrd -size b 0x0 3 Debugger accesses one half-word and one byte from the memory, displays three bytes. 3. mrd 0x0 3 Debugger accesses three words from the memory and displays three words. To read more than 64 bits of data, specify the number of data words along with the address. Data read is in multiples of access size. For example, to read 128 bits of data, run "mrd -size d <addr> 2" or "mrd -size w <addr> 4."
-value	Return a Tcl list of values, instead of displaying the result on the console.
-bin	Return data read from the target in binary format.
-file <file-name>	Write binary data read from the target to <file-name>.
-address-space <name>	Access specified memory space instead default memory space of current target. For Arm® DAP targets, address spaces DPR, APR, and AP<n> can be used to access DP registers, AP registers, and MEM-AP addresses respectively. For backwards compatibility, the -arm-dap and -arm-ap options can be used as shorthand for "-address-space APR" and "-address-space AP<n>" respectively. The APR address range is 0x0 - 0xffff, where the higher eight bits select an AP and the lower eight bits are the register address for that AP.
-unaligned-access	The memory address is not aligned to the access size before performing a read operation. Support for unaligned accesses is target architecture dependent. If this option is not specified, addresses are automatically aligned to access size.

Note(s)

- Select an APU target to access Arm DAP and MEM-AP address space.

Returns

Memory addresses and data in requested format, if successful. Error string, if the target memory cannot be read.

Examples

```
mrd 0x0
```

Read a word at 0x0.

```
mrd 0x0 10
```

Read 10 words at 0x0.

```
mrd -value 0x0 10
```

Read 10 words at 0x0 and return a Tcl list of values.

```
mrd -size b 0x1 3
```

Read three bytes at address 0x1.

```
mrd -size h 0x2 2
```

Read two half-words at address 0x2.

```
mrd -bin -file mem.bin 0 100
```

Read 100 words at address 0x0 and write the binary data to mem.bin.

```
mrd -address-space APR 0x100
```

Read APB-AP CSW on Zynq. The higher eight bits (0x1) select the APB-AP and the lower eight bits (0x0) are the address of CSW.

```
mrd -address-space APR 0x04
```

Read AHB-AP TAR on Zynq. The higher eight bits (0x0) select the AHB-AP and the lower eight bits (0x4) are the address of TAR.

```
mrd -address-space AP1 0x80090088
```

Read address 0x80090088 on DAP APB-AP. AP 1 selects the APB-AP. 0x80090088 on APB-AP corresponds to DBGDSCR of Cortex-A9#0, on Zynq.

```
mrd -address-space AP0 0xe000d000
```

Read address 0xe000d000 on DAP AHB-AP. AP 0 selects the AHB-AP. 0xe000d000 on AHB-AP corresponds to QSPI device on Zynq.

mwr

Memory write.

Syntax

```
mwr [options] <address> <values> [num]
```

Write <num> data values from list of <values> to active target memory address specified by <address>. If <num> is not specified, all the <values> from the list are written sequentially from the address specified by <address>. If <num> is greater than the size of the <values> list, the last word in the list is filled at the remaining address locations.

```
mwr [options] -bin -file <file-name> <address> [num]
```

Read <num> data values from a binary file and write to active target memory address specified by <address>. If <num> is not specified, all the data from the file is written sequentially from the address specified by <address>.

Options

Option	Description
-force	Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.
-bypass-cache-sync	Do not flush/invalidate CPU caches during memory write. Without this option, the debugger flushes/invalidates caches to make sure caches are in sync.
-size <access-size>	<access-size> can be one of the values below: b = Bytes accesses h = Half-word accesses w = Word accesses d = Double-word accesses Default access size is w. Address is aligned to access-size before writing to memory, if the '-unaligned-access' option is not used. If the target does not support double-word access, the debugger uses two word accesses. If number of data values to be written is more than 1, the debugger selects the appropriate access size. For example, 1. mwr -size b 0x0 {0x0 0x13 0x45 0x56} Debugger writes one word to the memory, combining four bytes. 2. mwr -size b 0x0 {0x0 0x13 0x45} Debugger writes one half-word and one byte to the memory, combining the three bytes. 3. mwr 0x0 {0x0 0x13 0x45} Debugger writes three words to the memory. To write more than 64 bits of data, specify the number of data words along with the address. Data written is in multiples of access size. For example, to write 128 bits of data, run "mwr -size d <addr> 2" or "mwr -size w <addr> 4."
-bin	Read binary data from a file and write it to the target address space.
-file <file-name>	File from which binary data is read, to write to the target address space.
-address-space <name>	Access specified memory space instead default memory space of current target. For Arm DAP targets, address spaces DPR, APR, and AP<n> can be used to access DP registers, AP registers, and MEM-AP addresses respectively. For backwards compatibility, the -arm-dap and -arm-ap options can be used as shorthand for "-address-space APR" and "-address-space AP<n>" respectively. The APR address range is 0x0 - 0xffff, where the higher eight bits select an AP and the lower eight bits are the register address for that AP.
-unaligned-accesses	Memory address is not aligned to access size before performing a write operation. Support for unaligned accesses is target architecture dependent. If this option is not specified, addresses are automatically aligned to access size.

Note(s)

- Select an APU target to access Arm DAP and MEM-AP address space.

Returns

Nothing, if successful. Error string, if the target memory cannot be written.

Examples

```
mwr 0x0 0x1234
```

Write 0x1234 to address 0x0.

```
mwr 0x0 {0x12 0x23 0x34 0x45}
```

Write four words from the list of values to address 0x0.

```
mwr 0x0 {0x12 0x23 0x34 0x45} 10
```

Write four words from the list of values to address 0x0 and fill the last word from the list at the remaining six address locations.

```
mwr -size b 0x1 {0x1 0x2 0x3} 3
```

Write three bytes from the list at address 0x1.

```
mwr -size h 0x2 {0x1234 0x5678} 2
```

Write two half-words from the list at address 0x2.

```
mwr -bin -file mem.bin 0 100
```

Read 100 words from binary file mem.bin and write the data at target address 0x0.

```
mwr -arm-dap 0x100 0x80000042
```

Write 0x80000042 to APB-AP CSW on Zynq. The higher eight bits (0x1) select the APB-AP and the lower eight bits (0x0) are the address of CSW.

```
mwr -arm-dap 0x04 0xf8000120
```

Write 0xf8000120 to AHB-AP TAR on Zynq. The higher eight bits (0x0) select the AHB-AP and the lower eight bits (0x4) are the address of TAR.

```
mwr -arm-ap 1 0x80090088 0x03186003
```

Write 0x03186003 to address 0x80090088 on DAP APB-AP. AP 1 selects the APB-AP. 0x80090088 on APB-AP corresponds to DBGDSCR of Cortex-A9#0, on Zynq.

```
mwr -arm-ap 0 0xe000d000 0x80020001
```

Write 0x80020001 to address 0xe000d000 on DAP AHB-AP. AP 0 selects the AHB-AP. 0xe000d000 on AHB-AP corresponds to the QSPI device on Zynq.

osa

Configure OS awareness for a symbol file.

Syntax

```
osa -file <file-name> [options]
```

Configure OS awareness for the symbol file <file-name> specified. If no symbol file is specified and only one symbol file exists in target's memory map, that symbol file is used. If no symbol file is specified and multiple symbol files exist in target's memory map, an error is thrown.

Options

Option	Description
-disable	Disable OS awareness for a symbol file. If this option is not specified, OS awareness is enabled.
-fast-exec	Enable fast process start. New processes are not tracked for debug and is not visible in the debug targets view.
-fast-step	Enable fast stepping. Only the current process is re-synced after stepping. All other processes are not resynced when this flag is turned on.

Note(s)

- The <fast-exec> and <fast-step> options are not valid with disable option.

Returns

Nothing, if the OSA is configured successfully. Error, if ambiguous options are specified.

Examples

```
osa -file <symbol-file> -fast-step -fast-exec
```

Enable OSA for <symbol-file> and turn on fast-exec and fast-step modes.

```
osa -disable -file <symbol-file>
```

Disable OSA for <symbol-file>.

memmap

Modify memory map.

Syntax

```
memmap <options>
```

Add/remove a memory map entry for the active target.

Options

Option	Description
-addr <memory-address>	Address of the memory region that should be added/removed from the target's memory map.
-alignment <bytes>	Force alignment during memory accesses for a memory region. If alignment is not specified, default alignment is chosen during memory accesses.
-size <memory-size>	Size of the memory region.
-flags <protection-flags>	Protection flags for the memory region. <protection-flags> can be a bitwise OR of the values below: 0x1 = Read access is allowed. 0x2 = Write access is allowed. 0x4 = Instruction fetch access is allowed. Default value of <protection-flags> is 0x3 (Read/Write Access).
-list	List the memory regions added to the active target's memory map.
-clear	Specify whether the memory region should be removed from the target's memory map.
-relocate-section-map <addr>	Relocate the address map of the program sections to <addr>. This option should be used when the code is self-relocating, so that the debugger can find the debug symbol info for the code. <addr> is the relative address, to which all the program sections are relocated.
-osa	Enable OS awareness for the symbol file. Fast process start and fast stepping options are turned off by default. These options can be enabled using the <osa> command. See "help osa" for more details.
-properties <dict>	Specify advanced memory map properties.
-meta-data <dict>	Specify meta-data of advanced memory map properties.

Note(s)

- Only the memory regions previously added through the memmap command can be removed.

Returns

Nothing, while setting the memory map. A list of memory maps when the -list option is used.

Examples

```
memmap -addr 0xfc000000 -size 0x1000 -flags 3
```

Add the memory region 0xfc000000 - 0xfc000fff to the target's memory map. Read/Write accesses are allowed to this region.

```
memmap -addr 0xfc000000 -clear
```

Remove the previously added memory region at 0xfc000000 from the target's memory map.

Target Download FPGA/BINARY

The following is a list of download commands:

- [dow](#)
- [verify](#)
- [fpga](#)

dow

Download ELF and binary file to target.

Syntax

```
dow [options] <file>
```

Download ELF file <file> to active target.

```
dow -data <file> <addr>
```

Download binary file <file> to active target address specified by <addr>.

Options

Option	Description
<code>-clear</code>	Clear uninitialized data (bss).
<code>-skip-tcm-clear</code>	When the R5 elfs are part of the PDI and use TCM, PLM initializes TCM before loading the elfs. Debugger does the same when the elfs are loaded through debugger, so that TCM banks are initialized properly. Use this option to skip initializing the TCM.
<code>-keepsym</code>	Keep previously downloaded ELFs in the list of symbol files. Default behavior is to clear the old symbol files while downloading an ELF.

Option	Description
<code>-force</code>	Overwrite access protection. By default, accesses to reserved and invalid address ranges are blocked.
<code>-bypass-cache-sync</code>	Do not flush/invalidate CPU caches during ELF download. Without this option, the debugger flushes/invalidates caches to make sure caches are in sync.
<code>-relocate-section-map <addr></code>	Relocate the address map of the program sections to <code><addr></code> . This option should be used when the code is self-relocating, so that the debugger can find debug symbol information for the code. <code><addr></code> is the relative address, to which all the program sections are relocated.
<code>-vaddr</code>	Use <code><vaddr></code> from the ELF program headers while downloading the ELF. This option is valid only for ELF files.

Returns

Nothing.

verify

Verify if ELF/binary file is downloaded correctly to target.

Syntax

```
verify [options] <file>
```

Verify if the ELF file specified by `<file>` is downloaded correctly to the active target.

```
verify -data <file> <addr>
```

Verify if the binary file specified by `<file>` is downloaded correctly to the active target address specified by `<addr>`.

Options

Option	Description
<code>-force</code>	Overwrite access protection. By default accesses to reserved and invalid address ranges are blocked.
<code>-vaddr</code>	Use <code><vaddr></code> from the ELF program headers while verifying the ELF data. This option is valid only for ELF files.

Returns

Nothing, if successful. Error string, if the memory address cannot be accessed or if there is a mismatch.

fpga

Configure FPGA.

Syntax

```
fpga <bitstream-file>
```

Configure FPGA with given bitstream.

```
fpga [options]
```

Configure FPGA with bitstream specified options, or read FPGA state.

Options

Option	Description
<code>-file <bitstream-file></code>	Specify file containing bitstream.
<code>-partial</code>	Configure FPGA without first clearing the current configuration. This option should be used while configuring partial bitstreams created before 2014.3 or any partial bitstreams in binary format.
<code>-no-revision-check</code>	Disable bitstream versus silicon revision compatibility check.
<code>-skip-compatibility-check</code>	Disable bitstream versus FPGA compatibility check.
<code>-state</code>	Return whether the FPGA is configured.
<code>-config-status</code>	Return configuration status.
<code>-ir-status</code>	Return IR capture status.
<code>-boot-status</code>	Return boot history status.
<code>-timer-status</code>	Return watchdog timer status.
<code>-cor0-status</code>	Return configuration option 0 status.
<code>-cor1-status</code>	Return configuration option 1 status.
<code>-wbstar-status</code>	Return warm boot start address status.

Note(s)

- If no target is selected or if the current target is not a supported FPGA, and only one supported FPGA is found in the targets list, this device is configured.

Returns

Depends on options used.

`-file`, `-partial`: Nothing, if FPGA is configured, or an error if the configuration failed.

One of the other options Configuration value.

Target Reset

The following is a list of reset commands:

- **rst**

rst

Target reset.

Syntax

```
rst [options]
```

Reset the active target.

Options

Option	Description
<code>-processor</code>	Reset the active processor target.
<code>-cores</code>	Reset the active processor group. This reset type is supported only on Zynq, Zynq UltraScale+ MPSoC, and Versal devices. A processor group is defined as a set of processor cores and on-chip peripherals like OCM.
<code>-dap</code>	Reset Arm DAP. This reset type is supported only with targets that represent Arm DAP. Examples of such targets are APU, RPU, PSU, and Versal.
<code>-system</code>	Reset the active system. This is the default reset.
<code>-srst</code>	Generate system reset for active target. With JTAG, this is done by generating a pulse on the SRST pin on the JTAG cable associated with the active target.
<code>-por</code>	Generate power on reset for active target. With JTAG, this is done by generating a pulse on the POR pin on the JTAG cable associated with the active target.
<code>-ps</code>	Generate PS only reset on Zynq UltraScale+ MPSoC. This is supported only through MicroBlaze PMU target.
<code>-stop</code>	Suspend cores after reset. If this option is not specified, the debugger chooses the default action, which is to resume the cores for <code>-system</code> , and suspend the cores for <code>-processor</code> , and <code>-cores</code> . This option is only supported with the <code>-processor</code> , <code>-cores</code> , and <code>-system</code> options.
<code>-start</code>	Resume the cores after reset. See the description of the <code>-stop</code> option for more details.
<code>-endianness <value></code>	Set the data endianness to <code><value></code> . The following values are supported: <code>le</code> - Little endian; <code>be</code> - Big endian. This option is supported with APU, RPU, A9, A53, and A72 targets. If this option is not specified, the current configuration is not changed.

Option	Description
-code-endianness <value>	Set the instruction endianness to <value>. The following values are supported: le - Little endian; be - Big endian. This option is supported with APU, RPU, A9, A53, and A72 targets. If this option is not specified, the current configuration is not changed.
-isa <isa-name>	Set ISA to <isa-name>. Supported isa-names are ARM/A32, A64, and Thumb. This option is supported with APU, RPU, A9, A53, and A72 targets. If this option is not specified, the current configuration is not changed.
-clear-registers	Clear CPU registers after a reset is triggered. This option is useful while triggering a reset after the device is powered up. Otherwise, debugger can end up reading invalid system addresses based on the register contents. Clearing the registers avoids unpredictable behavior. This option is supported for Arm targets, when used with '-processor' and '-cores'.
-type <reset type>	The following reset types are supported: core, cluster, cpu, dap, system, por, pmc-por, pmc-srst, ps-por, ps-srst, pl-por, and pl-srst. pmc-por, pmc-srst, ps-por, ps-srst, pl-por, and pl-srst are supported for Versal devices. Each of these reset types assert and deassert corresponding bits in RST_PS register of CRP module. pmc-por : RST_PS[PMC_POR] pmc-srst : RST_PS[PMC_SRST] ps-por : RST_PS[PS_POR] ps-srst : RST_PS[PS_SRST] pl-por : RST_PS[PL_POR] pl-srst : RST_PS[PL_SRST]

Note(s)

- For Versal devices, the default subsystem is activated through IPI channel5, before triggering the processor reset. This is needed because PLM does not activate the subsystem when PS ELF's are not part of the PDI. If the IPI channel is not enabled in the Vivado design, the subsystem cannot be activated. This causes runtime issues if PM API are used.

Returns

Nothing, if reset is successful. Error string, if reset is unsupported.

IPI commands to Versal PMC

The following is a list of ipi commands:

- [plm](#)

plm

PLM logging.

Syntax

```
plm <sub-command> [ options ]
```

Configure PLM log-level/log-memory, or copy/retrieve PLM log, based on the sub-command specified. The 'copy-debug-log' sub-command allows you to copy the PLM debug log to user memory. The 'set-debug-log' sub-command allows you to configure the memory for the PLM debug log. The 'set-log-level' sub-command allows you to configure the PLM log level. The 'log' command allows you to retrieve the PLM debug log. Type "help" followed by "plm sub-command", or "plm sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command. Refer to the help for the relevant sub-command for details.

Examples

Refer to the help for the relevant sub-command for details.

plm copy-debug-log

Copy PLM debug log.

Syntax

```
plm copy-debug-log <addr>
```

Copy PLM debug log from debug log buffer to user memory specified by <addr>.

Returns

Nothing, if successful. Error, otherwise.

Examples

```
plm copy-debug-log 0x0
```

Copy PLM debug log from the default log buffer to address 0x0.

plm set-debug-log

Configure PLM debug log memory.

Syntax

```
plm set-debug-log <addr> <size>
```

Specify the address and size of the memory which should be used for PLM debug log. By default, PMC RAM is used for PLM debug log.

Returns

Nothing, if successful. Error, otherwise.

Examples

```
plm set-debug-log 0x0 0x4000
```

Use the memory 0x0 - 0x3fff for PLM debug log.

plm set-log-level

Configure PLM log level.

Syntax

```
plm set-log-level <level>
```

Configure the PLM log level. This can be less than or equal to the level set during the compile time. The following levels are supported. 0x1 is for unconditional messages (DEBUG_PRINT_ALWAYS). 0x2 is for general debug messages (DEBUG_GENERAL). 0x3 is for more debug information (DEBUG_INFO). 0x4 is for detailed debug information (DEBUG_DETAILED).

Returns

Nothing, if successful. Error, otherwise.

Examples

```
plm set-log-level 0x1
```

Configure the log level to 1.

plm log

Retrieve the PLM log.

Syntax

```
plm log [options]
```

Retrieve the PLM log, and print it on the console, or a channel.

Options

Option	Description
<code>-handle <handle></code>	Specify the file handle to which the data should be redirected. If no file handle is given, data is printed on stdout.
<code>-log-mem-addr <addr></code>	Specify the memory address from which the PLM log should be retrieved. By default, the address and log size are obtained by triggering IPI commands to PLM. If PLM does not respond to IPI commands, default address 0xf2019000 is used. This option can be used to change default address. If either memory address or log size is specified, then the address and size are not retrieved from PLM. If only one of the address or size options is specified, default value is used for the other option. See below for description about log size.
<code>-log-size <size in bytes></code>	Specify the log buffer size. If this option is not specified, the default size of 1024 bytes is used, only when the log memory information cannot be retrieved from PLM.
<code>-slr <num></code>	Specify the slave slr number. If this option is not specified, the default is SLR0 (master plm). Valid slr range is from 0 to 3.

Returns

Nothing, if successful. Error, otherwise.

Examples

```
set fp [open test.log r]
plm log -handle $fp
```

Retrieve PLM debug log and write it to test.log.

```
plm log -slr 2
```

Retrieve PLM debug log from slave slr 2.

Target Breakpoints/Watchpoints

The following is a list of breakpoints commands:

- [bpadd](#)

- [bpremove](#)
- [benable](#)
- [bpdisable](#)
- [bplist](#)
- [bpstatus](#)

bpadd

Set a breakpoint/watchpoint.

Syntax

```
bpadd <options>
```

Set a software or hardware breakpoint at address, function or <file>:<line>, or set a read/write watchpoint, or set a cross-trigger breakpoint.

Options

Option	Description
-addr <breakpoint-address>	Specify the address at which the breakpoint should be set.
-file <file-name>	Specify the <file-name> in which the breakpoint should be set.
-line <line-number>	Specify the <line-number> within the file where the breakpoint should be set.
-type <breakpoint-type>	Specify the breakpoint type <breakpoint-type> can be one of the values below: auto = Auto-breakpoint type is chosen by the hw_server/TCF agent. This is the default type. hw = hardware breakpoint. sw = software breakpoint.
-mode <breakpoint-mode>	Specify the access mode that triggers the breakpoint. <breakpoint-mode> can be a bitwise OR of the values below: 0x1 is triggered by a read from the breakpoint location. 0x2 is triggered by a write to the breakpoint location. 0x4 is triggered by an instruction execution at the breakpoint location. This is the default for line and address breakpoints. 0x8 is triggered by a data change (not an explicit write) at the breakpoint location.
-enable <mode>	Specify initial enablement state of breakpoint. When <mode> is 0 the breakpoint is disabled, otherwise the breakpoint is enabled. The default is enabled.
-ct-input <list> -ct-output <list>	Specify input and output cross triggers. <list> is a list of numbers identifying the cross trigger pin. For Zynq 0-7 it is CTI for core 0, 8-15 is CTI for core 1, 16-23 is CTI ETB and TPIU, and 24-31 is CTI for FTM.
-skip-on-step <value>	Specify the trigger behaviour on stepping. This option is only applicable for cross trigger breakpoints and when DBGACK is used as breakpoint input. 0 = trigger every time core is stopped (default). 1 = suppress trigger on stepping over a code breakpoint. 2 = suppress trigger on any kind of stepping.

Option	Description
<code>-properties <dict></code>	Specify advanced breakpoint properties.
<code>-meta-data <dict></code>	Specify metadata of advanced breakpoint properties.
<code>-target-id <id></code>	Specify a target ID for which the breakpoint should be set. A breakpoint can be set for all the targets by specifying the <code><id></code> as 'all'. If this option is not used, the breakpoint is set for the active target selected through targets command. If there is no active target, the breakpoint is set for all targets.
<code>-temp</code>	The breakpoint is removed after it is triggered once.
<code>-skip-prologue</code>	For function breakpoints, the function prologue is skipped while planting the breakpoint.

Note(s)

- Breakpoints can be set in XSDB before connecting to hw_server/TCF agent. If there is an active target when a breakpoint is set, the breakpoint is enabled only for that active target. If there is no active target, the breakpoint is enabled for all the targets. The target-id option can be used to set a breakpoint for a specific target, or all targets. An address breakpoint or a file:line breakpoint can also be set without the options -addr, -file, or -line. For address breakpoints, specify the address as an argument, after all other options. For file:line breakpoints, specify the file name and line number in the format `<file>:<line>`, as an argument, after all other options.

Returns

Breakpoint id or an error if invalid target id is specified.

Examples

```
bpadd -addr 0x100000
```

Set a Breakpoint at address 0x100000. Breakpoint type is chosen by hw_server/TCF agent.

```
bpadd -addr &main
```

Set a function Breakpoint at main. Breakpoint type is chosen by hw_server/TCF agent.

```
bpadd -file test.c -line 23 -type hw
```

Set a hardware breakpoint at test.c:23.

```
bpadd -target-id all 0x100
```

Set a breakpoint for all targets, at address 0x100.

```
bpadd -target-id 2 test.c:23
```

Set a breakpoint for target 2, at line 23 in test.c.

```
bpadd -addr &fooVar -type hw -mode 0x3
```

Set a read/write watchpoint on variable fooVar.

```
bpadd -ct-input 0 -ct-output 8
```

Set a cross trigger to stop Zynq core 1 when core 0 stops.

bpremove

Remove breakpoints/watchpoints.

Syntax

```
bpremove <id-list> | -all
```

Remove the breakpoints/watchpoints specified by `<id-list>` or remove all the breakpoints when the `-all` option is used.

Options

Option	Description
<code>-all</code>	Remove all breakpoints.

Returns

Nothing, if the breakpoint is removed successfully. Error string, if the breakpoint specified by `<id>` is not set.

Examples

```
bpremove 0
```

Remove breakpoint 0.

```
bpremove 1 2
```

Remove breakpoints 1 and 2.

```
bpremove -all
```

Remove all breakpoints.

bopenable

Enable breakpoints/watchpoints.

Syntax

```
bopenable <id-list> | -all
```

Enable the breakpoints/watchpoints specified by `<id-list>` or enable all the breakpoints when `-all` option is used.

Options

Option	Description
<code>-all</code>	Enable all breakpoints.

Returns

Nothing, if the breakpoint is enabled successfully. Error string, if the breakpoint specified by `<id>` is not set.

Examples

```
bopenable 0
```

Enable breakpoint 0.

```
bopenable 1 2
```

Enable breakpoints 1 and 2.

```
bopenable -all
```

Enable all breakpoints.

bpdisable

Disable breakpoints/watchpoints.

Syntax

```
bpdisable <id-list> | -all
```

Disable the breakpoints/watchpoints specified by `<id-list>` or disable all the breakpoints when the `-all` option is used.

Options

Option	Description
-all	Disable all breakpoints.

Returns

Nothing, if the breakpoint is disabled successfully. Error string, if the breakpoint specified by `<id>` is not set.

Examples

```
bpdisable 0
```

Disable breakpoint 0.

```
bpdisable 1 2
```

Disable breakpoints 1 and 2.

```
bpdisable -all
```

Disable all breakpoints.

bplist

List breakpoints/watchpoints.

Syntax

```
bplist
```

List all the breakpoints/watchpoints along with brief status for each breakpoint and the target on which it is set.

Returns

List of breakpoints.

bpstatus

Print breakpoint/watchpoint status.

Syntax

```
bpstatus <id>
```

Print the status of a breakpoint/watchpoint specified by <`id`>. Status includes the target information for which the breakpoint is active and also the breakpoint hit count or error message.

Options

None.

Returns

Breakpoint status, if the breakpoint exists. Error string, if the breakpoint specified by <`id`> is not set.

Jtag UART

The following is a list of streams commands:

- [jtagterminal](#)
- [readjtaguart](#)

jtagterminal

Start/stop JTAG-based hyperterminal.

Syntax

```
jtagterminal [options]
```

Start/stop a JTAG-based hyperterminal to communicate with the Arm DCC or MDM UART interface.

Options

Option	Description
<code>-start</code>	Start the JTAG UART terminal. This is the default option.
<code>-stop</code>	Stop the JTAG UART terminal.
<code>-socket</code>	Return the socket port number instead of starting the terminal. External terminal programs can be used to connect to this port.

Note(s)

- Select a MDM or Arm/MicroBlaze processor target before running this command.

Returns

Socket port number.

readjtaguart

Start/stop reading from JTAG UART.

Syntax

```
readjtaguart [options]
```

Start/stop reading from the Arm® DCC or MDM UART TX interface. The JTAG UART output can be printed on stdout or redirected to a file.

Options

Option	Description
-start	Start reading the JTAG UART output.
-stop	Stop reading the JTAG UART output.
-handle <file-handle>	Specify the file handle to which the data should be redirected. If no file handle is given, data is printed on stdout.

Note(s)

- Select an MDM or Arm processor target before running this command.
- While running a script in non-interactive mode, the output from JTAG UART cannot be written to the log until "readjtaguart -stop" is used.

Returns

Nothing, if successful. Error string, if data cannot be read from the JTAG UART.

Examples

```
readjtaguart
```

Start reading from the JTAG UART and print the output on stdout.

```
set fp [open test.log w]; readjtaguart -start -handle $fp
```

Start reading from the JTAG UART and print the output to test.log.

```
readjtaguart -stop
```

Stop reading from the JTAG UART.

Miscellaneous

The following is a list of miscellaneous commands:

- [loadhw](#)
- [loadipxact](#)
- [unloadhw](#)
- [mdm_drwr](#)
- [mb_drwr](#)
- [mdm_drrd](#)
- [mb_drrd](#)
- [configparams](#)
- [version](#)
- [xsdbserver start](#)
- [xsdbserver stop](#)
- [xsdbserver disconnect](#)
- [xsdbserver version](#)

loadhw

Load a Vivado hardware design.

Syntax

```
loadhw [options]
```

Load a Vivado hardware design, and set the memory map for the current target. If the current target is a parent for a group of processors, the memory map is set for all of its child processors. If current target is a processor, the memory map is set for all the child processors of its parent. This command returns the hardware design object.

Options

Option	Description
-hw	Hardware design file.
-list	Return a list of open designs for the targets.

Option	Description
<code>-mem-ranges [list {start1 end1} {start2 end2}]</code>	List of memory ranges from which the memory map should be set. The memory map is not set for the addresses outside these ranges. If this option is not specified, the memory map is set for all the addresses in the hardware design.

Returns

Design object, if the hardware design is loaded and the memory map is set successfully. Error string, if the hardware design cannot be opened.

Examples

```
targets -filter {name =~ "APU"}; loadhw design.xsa
```

Load the hardware design named design.hdf and set the memory map for all the child processors of the APU target.

```
targets -filter {name =~ "xc7z045"}; loadhw design.xsa
```

Load the hardware design named design.hdf and set the memory map for all the child processors for which xc7z045 is the parent.

loadipxact

Load register definitions from ipxact file.

Syntax

```
loadipxact [options] [ipxact-xml]
```

Load memory mapped register definitions from an ipxact-xml file, or clear previously loaded definitions and return to built-in definitions, or return the XML file that is currently loaded.

Options

Option	Description
<code>-clear</code>	Clear definitions loaded from the ipxact file and return to built-in definitions.
<code>-list</code>	Return the ipxact file that is currently loaded.

Note(s)

- Select a target that supports physical memory accesses to load memory mapped register definitions. For example, APU, RPU, PSU, and Versal targets support physical memory accesses. Processor cores (A9, R5, A53, A72, etc.) support virtual memory accesses.

Returns

Nothing, if the ipxact file is loaded, or previously loaded definitions are cleared successfully. Error string, if load/clear fails. XML file path if -list option is used, and XML file is previously loaded.

Examples

```
loadipxact <xml-file>
```

Load register definitions from <xml-file>. This file should be in ipxact format.

```
loadipxact -clear
```

Clear previously loaded register definitions from an XML file, and return to built-in definitions.

```
loadipxact -list
```

Return the XML file that is currently loaded.

unloadhw

Unload a Vivado hardware design.

Syntax

```
unloadhw
```

Close the Vivado hardware design which was opened during the loadhw command, and clear the memory map for the current target. If the current target is a parent for a group of processors, the memory map is cleared for all its child processors. If the current target is a processor, the memory map is cleared for all the child processors of its parent. This command does not clear the memory map explicitly set by users through the memmap command.

Returns

Nothing.

mdm_drwr

Write to MDM debug register.

Syntax

```
mdm_drwr [options] <cmd> <data> <bitlen>
```

Write to MDM debug register. <cmd> is an 8-bit MDM command to access a debug register. <data> is the register value and <bitlen> is the register width.

Options

Option	Description
<code>-target-id <id></code>	Specify a target id representing the MicroBlaze debug module or MicroBlaze instance to access. If this option is not used and '-user' is not specified, the current target is used.
<code>-user <bscan number></code>	Specify user bscan port number.

Returns

Nothing, if successful.

Examples

```
mdm_drwr 8 0x40 8
```

Write to MDM break/reset control register.

mb_drwr

Write to MicroBlaze debug register.

Syntax

```
mb_drwr [options] <cmd> <data> <bitlen>
```

Write to the MicroBlaze debug register available on MDM. `<cmd>` is an 8-bit MDM command to access a debug register. `<data>` is the register value and `<bitlen>` is the register width.

Options

Option	Description
<code>-target-id <id></code>	Specify a target id representing a MicroBlaze instance to access. If this option is not used and -user is not specified, the current target is used.
<code>-user <bscan number></code>	Specify user bscan port number.
<code>-which <instance></code>	Specify MicroBlaze instance number.

Returns

Nothing, if successful.

Examples

```
mb_drwr 1 0x282 10
```

Write to MB control register.

mdm_drrd

Read from MDM debug register.

Syntax

```
mdm_drrd [options] <cmd> <bitlen>
```

Read an MDM debug register. `<cmd>` is an 8-bit MDM command to access a debug register and `<bitlen>` is the register width. Returns hex register value.

Options

Option	Description
<code>-target-id <id></code>	Specify a target id representing the MicroBlaze debug module or MicroBlaze instance to access. If this option is not used and
<code>-user</code> is not specified, the current target is used.	
<code>-user <bscan number></code>	Specify user bscan port number.

Returns

Register value, if successful.

Examples

```
mdm_drrd 0 32
```

Read XMDC ID register.

mb_drrd

Read from MicroBlaze Debug Register.

Syntax

```
mb_drrd [options] <cmd> <bitlen>
```

Read a MicroBlaze Debug Register available on MDM. cmd is 8-bit MDM command to access a Debug Register. bitlen is the register width. Returns hex register value.

Options

Option	Description
<code>-target-id <id></code>	Specify a target id representing MicroBlaze instance to access. If this option is not used and <code>-user</code> is not specified, then the current target is used.
<code>-user <bscan number></code>	Specify user bscan port number.
<code>-which <instance></code>	Specify MicroBlaze instance number.

Returns

Register value, if successful.

Examples

```
mb_drrd 3 28
```

Read MB Status Reg.

configparams

List, get, or set configuration parameters.

Syntax

```
configparams <options>
```

List the name and description for available configuration parameters. Configuration parameters can be global or connection specific, therefore the list of available configuration parameters and their value might change depending on the current connection.

```
configparams <options> <name>
```

Get configuration parameter value(s).

```
configparams <options> <name> <value>
```

Set configuration parameter value.

Options

Option	Description
<code>-all</code>	Include values for all contexts in result.
<code>-context [context]</code>	Specify context of value to get or set. The default context is "", which represents the global default. Not all options support context-specific values.

Option	Description
<code>-target-id <id></code>	Specify target id or value to get or set. This is an alternative to the <code>-context</code> option.

Returns

Depends on the arguments specified.

`<none>`: List of parameters and description of each parameter.

`<parameter name>`: Parameter value or error, if unsupported parameter is specified.

`<parameter name> <parameter value>`: Nothing if the value is set, or error, if the unsupported parameter is specified.

Examples

```
configparams force-mem-accesses 1
```

Disable access protection for the `<dow>`, `<mrd>`, and `<mwr>` commands.

```
configparams vitis-launch-timeout 100
```

Change the Vitis launch timeout to 100 seconds (used for running Vitis batch mode commands).

version

Get Vitis or hw_server version.

Syntax

```
version [options]
```

Get Vitis or hw_server version. When no option is specified, the Vitis build version is returned.

Options

Option	Description
<code>-server</code>	Get the hw_server build version for the active connection.

Returns

Vitis or hw_server version, on success. Error string, if server version is requested when there is no connection.

xsdbserver start

Start XSDB command server.

Syntax

```
xsdbserver start [options]
```

Start XSDB command server listener. The XSDB command server allows external processes to connect to XSDB to evaluate commands. The XSDB server reads commands from the connected socket one line at a time. After evaluation, a line is sent back starting with 'okay' or 'error' followed by the result or error as a backslash quoted string.

Options

Option	Description
-host <addr>	Limits the network interface on which to listen for incoming connections.
-port <port>	Specifies port to listen on. If this option is not specified, or if the port is zero, a dynamically allocated port number is used.

Returns

Server details are displayed on the console if the server is started successfully. Error string, if a server has been already started.

Examples

```
xsdbserver start
```

Start XSDB server listener using dynamically allocated port.

```
xsdbserver start -host localhost -port 2000
```

Start XSDB server listener using port 2000 and only allow incoming connections on this host.

xsdbserver stop

Stop XSDB command server.

Syntax

```
xsdbserver stop
```

Stop XSDB command server listener and disconnect connected client if any.

Returns

Nothing, if the server is closed successfully. Error string, if the server has not been started already.

xsdbserver disconnect

Disconnect active XSDB server connection.

Syntax

```
xsdbserver disconnect
```

Disconnect current XSDB server connection.

Returns

Nothing, if the connection is closed. Error string, if there is no active connection.

xsdbserver version

Return XSDB command server version.

Syntax

```
xsdbserver version
```

Return XSDB command server protocol version.

Returns

Server version if there is an active connection. Error string, if there is no active connection.

JTAG Access

The following is a list of jtag commands:

- [jtag targets](#)
- [jtag sequence](#)
- [jtag device_properties](#)
- [jtag lock](#)
- [jtag unlock](#)

- [jtag claim](#)
- [jtag disclaim](#)
- [jtag frequency](#)
- [jtag skew](#)
- [jtag servers](#)

jtag targets

List JTAG targets or switch between JTAG targets.

Syntax

```
jtag targets
```

List available JTAG targets.

```
jtag targets <target id>
```

Select <target id> as active JTAG target.

Options

Option	Description
-set	Set current target to entry single entry in list. This is useful in combination with -filter option. An error is generated if list is empty or contains more than one entry.
-regexp	Use regexp for filter matching.
-nocase	Use case insensitive filter matching.
-filter <filter-expression>	Specify filter expression to control that targets are included in list based on its properties. Filter expressions are similar to Tcl expr syntax. Target properties are referenced by name, while Tcl variables are accessed using the \$ syntax, string must be quoted. Operators ==, !=, <=, >=, <, >, &&, , and () are supported. These operators behave like Tcl expr operators. String matching operator =~ and !~ match LHS string with RHS pattern using either regexp or string match.
-target-properties	Returns a Tcl list of dictionaries containing target properties.
-open	Open all targets in list. List can be shorted by specifying target-ids and using filters.
-close	Close all targets in list. List can be shorted by specifying target-ids and using filters.
-timeout <sec>	Poll until the targets specified by filter option are found on the scan chain, or until timeout. This option is valid only with filter option. The timeout value is in seconds. Default timeout is three seconds.

Returns

The return value depends on the options used.

<none>: JTAG targets list when no options are used.

-filter: Filtered JTAG targets list.

-target-properties: Tcl list consisting of JTAG target properties.

An error is returned when JTAG target selection fails.

Examples

```
jtag targets
```

List all targets.

```
jtag targets -filter {name == "arm_dap"}
```

List targets with name "arm_dap."

```
jtag targets 2
```

Set target with id 2 as the current target.

```
jtag targets -set -filter {name =~ "arm*"}
```

Set current target to target with name starting with "arm."

```
jtag targets -set -filter {level == 0}
```

List JTAG cables.

jtag sequence

Create JTAG sequence object.

Syntax

```
jtag sequence
```

Create JTAG sequence object. DESCRIPTION The `jtag sequence` command creates a new sequence object. After creation the sequence is empty. The following sequence object commands are available:

```
sequence state new-state [count]
```

Move JTAG state machine to `<new-state>` and then generate `<count>` JTAG clocks. If `<clock>` is given and `<new-state>` is not a looping state (RESET, IDLE, IRSHIFT, IRPAUSE, DRSHIFT or DRPAUSE), the state machine moves towards RESET state.

```
sequence irshift [options] [bits [data]]
```

sequence drshift [options] bits [data] Shift data in IRSHIFT or DRSHIFT state. Data is either given as the last argument or if `-tdi` option is given then data is all zeros or all ones depending on the argument given to `-tdi`. The `<bits>` and `<data>` arguments are not used for irshift when the `-register` option is specified. Available options:

- `-register <name>` Select instruction register by name. This option is only supported for irshift.
- `-tdi <value>` TDI value to use for all clocks in SHIFT state.
- `-binary` Format of `<data>` is binary, for example data from a file or from binary format.
- `-integer` Format of `<data>` is an integer. The least significant bit of data is shifted first.
- `-bits` Format of `<data>` is a binary text string. The first bit in the string is shifted first.
- `-hex` Format of `<data>` is a hexadecimal text string. The least significant bit of the first byte in the string is shifted first.
- `-capture` Capture TDO data during shift and return from sequence run command.
- `-state <new-state>` State to enter after shift is complete. The default is RESET.

```
sequence delay usec
```

Generate the delay between sequence commands. No JTAG clocks are generated during the delay. The delay is guaranteed to be at least `<usec>` microseconds, but can be longer for cables that do not support delays without generating JTAG clocks.

```
sequence get_pin pin
```

Get value of `<pin>`. Supported pins are cable specific.

```
sequence set_pin pin value
```

Set value of `<pin>` to `<value>`. Supported pins are cable specific.

```
sequence atomic enable
```

Set or clear atomic sequences. This is useful to creating sequences that are guaranteed to run with precise timing or fail. Atomic sequences should be as short as possible to minimize the risk of failure.

```
sequence run [options]
```

Run JTAG operations in sequence for the currently selected jtag target. This command is return the result from shift commands using `-capture` option and from `get_pin` commands.

Available options are listed as follow-

- **binary** Format return value(s) as binary. The first bit shifted out is the least significant bit in the first byte returned.
- **-integer** Format return values(s) as integer. The first bit shifted out is the least significant bit of the integer.
- **-bits** Format return value(s) as binary text string. The first bit shifted out is the first character in the string.
- **-hex** Format return value(s) as hexadecimal text string. The first bit shifted out is the least significant bit of the first byte of the in the string.
- **-single** Combine all return values as a single piece of data. Without this option the return value is a list with one entry for every shift with **-capture** and every **get_pin**.

```
sequence clear
```

Remove all commands from sequence.

```
sequence delete
```

Delete sequence.

Returns

JTAG sequence object.

Examples

```
set seqname [jtag sequence]
$seqname state RESET
$seqname drshift -capture -tdi 0 256
set result [$seqname run]
$seqname delete
```

jtag device_properties

Get/set device properties.

Syntax

```
jtag device_properties idcode
```

Get JTAG device properties associated with `<idcode>`.

```
jtag device_properties key value ...
```

Set JTAG device properties.

Returns

Jtag device properties for the given idcode, or nothing, if the idcode is unknown.

Examples

```
jtag device_properties 0x4ba00477
```

Return Tcl dict containing device properties for idcode 0x4ba00477.

```
jtag device_properties {idcode 0x4ba00477 mask 0xffffffff name dap irlen 4}
```

Set device properties for idcode 0x4ba00477.

jtag lock

Lock JTAG scan chain.

Syntax

```
jtag lock [timeout]
```

Lock JTAG scan chain containing current JTAG target. DESCRIPTION Wait for scan chain lock to be available and then lock it. If <timeout> is specified the wait time is limited to <timeout> milliseconds. The JTAG lock prevents other clients from performing any JTAG shifts or state changes on the scan chain. Other scan chains can be used in parallel. The jtag run_sequence command ensures that all commands in the sequence are performed in order so the use of jtag lock is only needed when multiple jtag run_sequence commands needs to be done without interruption.

Note(s)

- A client should avoid locking more than one scan chain as this can cause dead-lock.

Returns

Nothing.

jtag unlock

Unlock JTAG scan chain.

Syntax

```
jtag unlock
```

Unlock JTAG scan chain containing current JTAG target.

Returns

Nothing.

jtag claim

Claim JTAG device.

Syntax

```
jtag claim <mask>
```

Set claim mask for current JTAG device.

DESCRIPTION- This command attempts to set the claim mask for the current JTAG device. If any set bits in <mask> are already set in the claim mask then this command returns error-

```
"already claimed".
```

The claim mask allow clients to negotiate control over JTAG devices. This is different from jtag lock in that 1) it is specific to a device in the scan chain, and 2) any clients can perform JTAG operations while the claim is in effect.

Note(s)

- Currently claim is used to disable the hw_server debugger from controlling microprocessors on Arm DAP devices and FPGA devices containing MicroBlaze processors.

Returns

Nothing.

jtag disclaim

Disclaim JTAG device.

Syntax

```
jtag disclaim <mask>
```

Clear claim mask for current JTAG device.

Returns

Nothing.

jtag frequency

Get/set JTAG frequency.

Syntax

```
jtag frequency
```

Get JTAG clock frequency for current scan chain.

```
jtag frequency -list
```

Get list of supported JTAG clock frequencies for current scan chain.

```
jtag frequency <frequency>
```

Set JTAG clock frequency for current scan chain. This frequency is persistent as long as the `hw_server` is running, and is reset to the default value when a new `hw_server` is started.

Returns

Current JTAG frequency, if no arguments are specified, or if JTAG frequency is successfully set. Supported JTAG frequencies, if `-list` option is used. Error string, if invalid frequency is specified or frequency cannot be set.

jtag skew

Get/set JTAG skew.

Syntax

```
jtag skew
```

Get JTAG clock skew for current scan chain.

```
jtag skew <clock-skew>
```

Set JTAG clock skew for current scan chain.

Note(s)

- Clock skew property is not supported by some JTAG cables.

Returns

Current JTAG clock skew, if no arguments are specified, or if JTAG skew is successfully set. Error string, if invalid skew is specified or skew cannot be set.

jtag servers

List, open or close JTAG servers.

Syntax

```
jtag servers [options]
```

List, open, and close JTAG servers. JTAG servers are used to implement support for different types of JTAG cables. An open JTAG server will enumerate or connect to available JTAG ports.

Options

Option	Description
-list	List opened servers. This is the default if no other option is given.
-format	Return the format of each supported server string.
-open <server>	Specifies server to open.
-close <server>	Specifies server to close.

Returns

Depends on the options specified.

<none>, -list: List of open JTAG servers.

-format: List of supported JTAG servers.

-close: Nothing if the server is closed, or an error string, if invalid server is specified.

Examples

```
jtag servers
```

List opened servers and number of associated ports.

```
jtag servers -open xilinx-xvc:localhost:10200
```

Connect to XVC server on host localhost port 10200.

```
jtag servers -close xilinx-xvc:localhost:10200
```

Close XVC server for host localhost port 10200.

Target File System

The following is a list of tfile commands:

- [tfile open](#)
- [tfile close](#)
- [tfile read](#)
- [tfile write](#)
- [tfile stat](#)
- [tfile lstat](#)
- [tfile fstat](#)
- [tfile setstat](#)
- [tfile fsetstat](#)
- [tfile remove](#)
- [tfile rmdir](#)
- [tfile mkdir](#)
- [tfile realpath](#)
- [tfile rename](#)
- [tfile readlink](#)
- [tfile symlink](#)
- [tfile opendir](#)
- [tfile readdir](#)
- [tfile copy](#)
- [tfile user](#)
- [tfile roots](#)
- [tfile ls](#)

tfile open

Open file.

Syntax

```
tfile open <path>
```

Open specified file.

Returns

File handle.

tfile close

Close file handle.

Syntax

```
tfile close <handle>
```

Close specified file handle.

Returns

Nothing.

tfile read

Read file handle.

Syntax

```
tfile read <handle>
```

Read from specified file handle.

Options

Option	Description
-offset <seek>	File offset to read from.

Returns

Read data.

tfile write

Write file handle.

Syntax

```
tfile write <handle>
```

Write to specified file handle.

Options

Option	Description
-offset <seek>	File offset to write to.

Returns

Nothing.

tfile stat

Get file attributes from path.

Syntax

```
tfile stat <handle>
```

Get file attributes for <path>.

Returns

File attributes.

tfile lstat

Get link file attributes from path.

Syntax

```
tfile lstat <path>
```

Get link file attributes for <path>.

Returns

Link file attributes.

tfile fstat

Get file attributes from handle.

Syntax

```
tfile fstat <handle>
```

Get file attributes for <handle>.

Returns

File attributes.

tfile setstat

Set file attributes for path.

Syntax

```
tfile setstat <path> <attributes>
```

Set file attributes for <path>.

Returns

File attributes.

tfile fsetstat

Set file attributes for handle.

Syntax

```
tfile fsetstat <handle> <attributes>
```

Set file attributes for <handle>.

Returns

File attributes.

tfile remove

Remove path.

Syntax

```
tfile remove <path>
```

Remove <path>.

Returns

Nothing.

tfile rmdir

Remove directory.

Syntax

```
tfile rmdir <path>
```

Remove directory <path>.

Returns

Nothing.

tfile mkdir

Create directory.

Syntax

```
tfile mkdir <path>
```

Make directory <path>.

Returns

Nothing.

tfile realpath

Get real path.

Syntax

```
tfile realpath <path>
```

Get real path of <path>.

Returns

Real path.

tfile rename

Rename path.

Syntax

```
tfile rename <old path> <new path>
```

Rename file or directory.

Returns

Nothing.

tfile readlink

Read symbolic link.

Syntax

```
tfile readlink <path>
```

Read link file.

Returns

Target path.

tfile symlink

Create symbolic link.

Syntax

```
tfile symlink <old path> <new path>
```

Symlink file or directory.

Returns

Nothing.

tfile opendir

Open directory.

Syntax

```
tfile opendir <path>
```

Open directory <path>.

Returns

File handle.

tfile readdir

Read directory.

Syntax

```
tfile readdir <file handle>
```

Read directory.

Returns

File handle.

tfile copy

Copy target file.

Syntax

```
tfile copy <src> <dest>
```

Copy file <src> to <dest>.

Returns

Copy file locally on target.

tfile user

Get user attributes.

Syntax

```
tfile user
```

Get user attributes.

Returns

User information.

tfile roots

Get file system roots.

Syntax

```
tfile roots
```

Get file system roots.

Returns

List of file system roots.

tfile ls

List directory contents.

Syntax

```
tfile ls <path>
```

List directory contents.

Returns

Directory contents.

SVF Operations

The following is a list of svf commands:

- [svf config](#)
- [svf generate](#)

- [svf mwr](#)
- [svf dow](#)
- [svf stop](#)
- [svf con](#)
- [svf delay](#)
- [svf rst](#)

svf config

Configure options for SVF file.

Syntax

```
svf config [options]
```

Configure and generate SVF file.

Options

Option	Description
-scan-chain <list of idcode-irlength pairs>	List of idcode-irlength pairs. This can be obtained from xsdb command - jtag targets
-device-index <index>	This is used to select device in the jtag scan chain.
-cpu-index <processor core>	Specify the cpu-index to generate the SVF file. For A53#0 - A53#3 on ZynqMP, use cpu-index 0 - 3 For R5#0 - R5#1 on ZynqMP, use cpu-index 4 - 5 For A9#0 - A9#1 on Zynq, use cpu-index 0 - 1 If multiple MicroBlaze processors are connected to MDM, select the specific MicroBlaze index for execution.
-out <filename>	Output SVF file.
-delay <tcks>	Delay in ticks between AP writes.
-linkdap	Generate SVF for linking DAP to the jtag chain for ZynqMP Silicon versions 2.0 and above.
-bscan <user port>	This is used to specify user bscan port to which MDM is connected.
-mb-chunksize <size in bytes>	This option is used to specify the chunk size in bytes for each transaction while downloading. Supported only for MicroBlaze processors.
-exec-mode	Execution mode for Arm v8 cores. Supported modes are a32 (v8 core is set up in 32-bit mode) and a64 (v8 core is set up in 64-bit mode).

Returns

Nothing.

Examples

```
svf config -scan-chain {0x14738093 12 0x5ba00477 4} -device-index 1 -cpu-index 0 -out "test.svf"
```

This creates a SVF file with name test.svf for core A53#0

```
svf config -scan-chain {0x14738093 12 0x5ba00477 4} -device-index 0 -bscan pmu -cpu-index 0 -out "test.svf"
```

This creates a SVF file with name test.svf for PMU MB

```
svf config -scan-chain {0x23651093 6} -device-index 0 -cpu-index 0 -bscan user1 -out "test.svf"
```

This creates a SVF file with name test.svf for MB connected to MDM on bscan USER1

svf generate

Generate recorded SVF file.

Syntax

```
svf generate
```

Generate SVF file in the path specified in the config command.

Options

None.

Returns

If successful, this command returns nothing. Otherwise it returns an error.

Examples

```
svf generate
```

svf mwr

Record memory write to SVF file.

Syntax

```
svf mwr <address> <value>
```

Write <value> to the memory address specified by <address>.

Options

None.

Returns

If successful, this command returns nothing. Otherwise it returns an error.

Examples

```
svf mwr 0xfffff0000 0x14000000
```

svf dow

Record elf download to SVF file.

Syntax

```
svf dow <elf file>
```

Record downloading of elf file <elf file> to the memory.

```
svf dow -data <file> <addr>
```

Record downloading of binary file <file> to the memory.

Options

None.

Returns

If successful, this command returns nothing. Otherwise it returns an error.

Examples

```
svf dow "fsbl.elf"
```

Record downloading of elf file fsbl.elf.

```
svf dow -data "data.bin" 0x1000
```

Record downloading of binary file data.bin to the address 0x1000.

svf stop

Record stopping of core to SVF file.

Syntax

```
svf stop
```

Record suspending execution of current target to SVF file.

Options

None.

Returns

Nothing.

Examples

```
svf stop
```

svf con

Record resuming of core to SVF file.

Syntax

```
svf con
```

Record resuming the execution of active target to SVF file.

Options

None.

Returns

Nothing.

Examples

```
svf con
```

svf delay

Record delay in tcks to SVF file.

Syntax

```
svf delay <delay in tcks>
```

Record delay in tcks to SVF file.

Options

None.

Returns

Nothing.

Examples

```
svf delay 1000
```

Delay of 1000 tcks is added to the SVF file.

svf rst

Reset

Syntax

```
svf rst
```

System Reset

Options

None.

Returns

If successful, this command returns nothing. Otherwise it returns an error.

Examples

```
svf rst
```

Device Configuration System

The following is a list of device commands:

- [device program](#)
- [device status](#)

- [device authjtag](#)

device program

Program PDI/BIT.

Syntax

```
device program <file>
```

Program PDI or BIT file into the device.

Note(s)

- If no target is selected or if the current target is not a configurable device, and only one supported device is found in the targets list, then this device will be configured. Otherwise, users will have to select a device using targets command.
- device program command is currently supported for Versal devices only. Other devices will be supported in future releases.
- For Versal devices, users can run "plm log" to retrieve plm log from memory.

Returns

Nothing, if device is configured, or an error if the configuration failed.

device status

Return JTAG register status.

Syntax

```
device status [options] <jtag-register-name>
```

Return device JTAG Register status, or list of available registers if no name is given.

Options

Option	Description
-jreg-name <jtag-register-name>	Specify jtag register name to read. This is the default option, so register name can be directly specified as an argument without using this option.
-jtag-target <jtag-target-id>	Specify jtag target id to use instead of the current target. This is primarily used when there isn't a valid target option.
-hex	Format the return data in hexadecimal.
-slr <slr-index>	Select the SLR from which to read the register (default SLR 0).

Returns

Status report.

device authjtag

Secure debug BIN.

Syntax

```
device authjtag <file>
```

Unlock device for secure debug.

Options

Option	Description
<code>-jtag-target <jtag-target-id></code>	Specify jtag target id to use instead of the current target. This is primarily used when there isn't a valid target option.

Note(s)

- If no target is selected or if the current target is not a configurable device, and only one supported device is found in the targets list, then this device will be configured. Otherwise, users will have to select a device using targets command.
- device authjtag command is currently supported for Versal devices only.

Returns

Nothing, if secure debug is successful, or an error if failed.

STAPL Operations

The following is a list of stapl commands:

- [stapl config](#)
- [stapl start](#)
- [stapl stop](#)

stapl config

Configure stapl target.

Syntax

```
stapl config <options>
```

Create a hw_target (jtag chain) and add all the hw_devices given in the scan-chain list to the hw_target. It also configures the stapl output file where the stapl data is recorded.

Options

Option	Description
-out <filepath>	Output file path. Only one of the -out and -handle options should be used. If the -out option is provided, the file will be explicitly opened in a+ mode.
-handle <filehandle>	File handle returned by open command for output. Only one of the -out and -handle options should be used.
-scan-chain <list-of-dicts>	List of devices in the scan-chain. Each list element must be a dict of device properties in the format {name <string> idcode <int> irlen <int> idcode2 <int> mask <int>}. For example: [list [dict create name <device1_name> idcode <idcode> irlen <irlen> idcode2 <idcode2> mask <mask>] [dict create name <device2_name> idcode <idcode> irlen <irlen> idcode2 <idcode2> mask <mask>]] The order of devices specified with scan-chain option should match the order of devices on the physical hardware where the stapl file is played back. Only one of the -scan-chain and -part options should be used.
-part <device-name list>	List of part names of the AMD devices to add to the scan-chain. This option works only with AMD devices. This option can be used instead of the -scan-chain option.
-checksum	Calculate stapl-data CRC and append it to the stapl file. If not specified, CRC 0 is appended.

Note(s)

- For AMD devices, if the device_name or idcode is specified in the scan-chain information, the other parameters are optional. All the JTAG TAPs are added automatically to the scan-chain for AMD devices.

Returns

None.

Examples

```
stapl config -handle $fp -scan-chain [list [dict create name xcvc1902
idcode 0 irlen 0 idcode2 0 mask 0] [dict create name xcvm1802 idcode 0
irlen 0 idcode2 0 mask 0]]
```

Add xcvc1902 and xcvm1802 devices to scan-chain and use the file handle returned by Tcl open command, to record stapl commands.

```
stapl config -out mystapl.stapl -scan-chain [list [dict create name xcvc1902 idcode 0 irlen 0 idcode2 0 mask 0] [dict create name xcvm1802 idcode 0 irlen 0 idcode2 0 mask 0]]
```

Same as the previous example, but using the stapl file path as input,

```
instead of the file handle returned by Tcl open command.  
stapl config -out mystapl.stapl -part xcvc1902
```

Add xcvc1902 device to scan-chain, using -part option.

```
stapl config -out mystapl.stapl -scan-chain [list [dict create idcode 0x14CA8093 idcode2 1]]
```

Same as previous example, but specifying idcode and idcode2, instead of the part name.

```
stapl config -out mystapl.stapl -part [list xcvc1902 xcvm1802]
```

Add xcvc1902 and xcvm1802 devices to scan-chain, using the -part option.

```
connect  
stapl config -out mystapl.stapl -scan-chain [list [dict create name xcvc1902 idcode 0 irlen 0 idcode2 0 mask 0]]  
jtag targets -set -filter {name == "xcvc1902"}  
stapl start  
device program <pdipath>  
stapl stop
```

The above example demonstrate the correct order for creating a stapl file for a single device on a stapl target.

```
connect  
stapl config -out mystapl.stapl -scan-chain [list [dict create name xcvc1902 idcode 0 irlen 0 idcode2 0 mask 0] [dict create xcvm1802 idcode 0 irlen 0 idcode2 0 mask 0]]  
jtag targets -set -filter {name == "xcvc1902"}  
targets -set -filter {jtag_device_name == "xcvc1902"}  
stapl start  
device program <pdipath>  
jtag targets -set -filter {name == "xcvm1802"}  
targets -set -filter {jtag_device_name == "xcvm1802"}  
stapl start  
device program <pdipath>  
stapl stop
```

The above example demonstrate the correct order for creating a stapl file for multiple devices on a stapl target.

stapl start

Start stapl recording.

Syntax

```
stapl start
```

Start stapl recording.

Options

None.

Note(s)

- It is mandatory to call 'stapl start' before programming each device on the scan-chain, and call 'stapl stop' after programming all the devices to generate stapl data properly.

Returns

None.

stapl stop

Stop stapl recording.

Syntax

```
stapl stop
```

Stop stapl recording.

Options

None.

Note(s)

- It is mandatory to call 'stapl start' before programming each device on the scan-chain, and call 'stapl stop' after programming all the devices to generate stapl data properly.

Returns

None.

Vitis Projects

The following is a list of projects commands:

- [openhw](#)
- [closehw](#)
- [getaddrmap](#)
- [getperipherals](#)
- [getprocessors](#)
- [repo](#)
- [lscript](#)
- [platform](#)
- [domain](#)
- [bsp](#)
- [library](#)
- [checkvalidrmxsa](#)
- [isstaticxsa](#)
- [ishwexpandable](#)
- [createdts](#)
- [setws](#)
- [getws](#)
- [app](#)
- [sysproj](#)
- [importprojects](#)
- [importsources](#)
- [toolchain](#)

openhw

Open a hardware design.

Syntax

```
openhw <hw-proj | xsa file>
```

Open a hardware design exported from Vivado. XSA file exported from Vivado, or the hardware project created using 'createhw' command can be passed as argument.

Options

None.

Returns

If successful, this command returns nothing. Otherwise it returns an error.

Examples

```
openhw ZC702_hw_platform
```

Open the hardware project ZC702_hw_platform.

```
openhw /tmp/wrk/hw1/system.xsa
```

Open the hardware project corresponding to the system.xsa.

closehw

Close a hardware design.

Syntax

```
closehw <hw project | xsa file>
```

Close a hardware design that was opened using 'openhw' command. XSA file exported from Vivado, or the hardware project created using 'createhw' command can be passed as argument.

Options

None.

Returns

If successful, this command returns nothing. Otherwise it returns an error.

Examples

```
closehw ZC702_hw_platform
```

Close the hardware project ZC702_hw_platform.

```
closehw /tmp/wrk/hw1/system.xsa
```

Close the hardware project corresponding to the system.xsa.

getaddrmap

Get the address ranges of IP connected to processor.

Syntax

```
getaddrmap <hw spec file> <processor-instance>
```

Return the address ranges of all the IP connected to the processor in a tabular format, along with details like size and access flags of all IP.

Options

None.

Returns

If successful, this command returns the output of IPs and ranges. Otherwise it returns an error.

Examples

```
getaddrmap system.xsa ps7_cortexa9_0
```

Return the address map of peripherals connected to ps7_cortexa9_0. system.xsa is the hw specification file exported from Vivado.

getperipherals

Get a list of all peripherals in the HW design

Syntax

```
getperipherals <xsa> <processor-instance>
```

Return the list of all the peripherals in the hardware design, along with version and type. If [processor-instance] is specified, return only a list of slave peripherals connected to that processor.

Options

None.

Returns

If successful, this command returns the list of peripherals. Otherwise it returns an error.

Examples

```
getperipherals system.xsa
```

Return a list of peripherals in the hardware design.

```
getperipherals system.xsa ps7_cortexa9_0
```

Return a list of peripherals connected to processor ps7_cortexa9_0 in the hardware design.

getprocessors

Get a list of all processors in the hardware design.

Syntax

```
getprocessors <xsa>
```

Return the list of all the processors in the hardware design

Options

None.

Returns

If successful, this command returns the list of processors. Otherwise, it returns an error.

Examples

```
getprocessors system.xsa
```

Return a list of processors in the hardware design.

repo

Get, set, or modify software repositories

Syntax

```
repo [OPTIONS]
```

Get/set the software repositories path currently used. This command is used to scan the repositories, to get the list of OS/libs/drivers/apps from repository.

Options

Option	Description
<code>-set <path-list></code>	Set the repository path and load all the software cores available. Multiple repository paths can be specified as Tcl list.
<code>-get</code>	Get the repository path(s).
<code>-scan</code>	Scan the repositories. Used this option to scan the repositories, when some changes are done.
<code>-os</code>	Return a list of all the OS from the repositories.
<code>-libs</code>	Return a list of all the libs from the repositories.
<code>-drivers</code>	Return a list of all the drivers from the repositories.
<code>-apps</code>	Return a list of all the applications from repositories along with the following details. Supported processor - Processors for which the application can be built. Supported OS - OS for which the application can be built. Platform required - Indicates whether a platform is required to create the application. AIE applications need a platform while other applications can be created using a platform or xsa.
<code>-add-platforms <platforms directory></code>	Add the specified directory to the platform repository.
<code>-remove-platforms-dir <platforms directory></code>	Remove the specified directory from the platform repository.

Returns

Depends on the OPTIONS specified.

`-scan`, `-set`: Returns nothing.

`-get`: Returns the current repository path.

`-os`, `-libs`, `-drivers`, `-apps`: Returns the list of OS/libs/drivers/apps respectively.

Examples

```
repo -set <repo-path>
```

Set the repository path to the path specified by `<repo-path>`.

```
repo -os
```

Return a list of OS from the repo.

```
repo -libs
```

Return a list of libraries from the repo.

Iscript

Create linker script.

Syntax

```
lscript <sub-command> [options]
```

Create a linkerscript, or perform various other operations on the linker script, based on the sub-command specified. Following sub-commands are supported. memory - List of the memories supported by the active domain. section - Lists and edit the sections available. def-mem - Returns default memory for the section type. generate - Generate a linker script. Type "help" followed by "lscript sub-command", or "lscript sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command. Refer to the sub-command help for details.

Examples

Refer to the sub-command help for details.

Iscript memory

List supported memory.

Syntax

```
lscript memory [options]
```

List of the memories supported by the active domain.

Options

Option	Description
-app <application-name>	Name of application from workspace.
-supported-mem	Returns supported memory regions for each section.

Returns

List of the memories supported by the active domain in tabular format.

Examples

```
lscript memory
```

This command returns the list of memories available in the active domain.

```
lscript memory -app <application-name>
```

Returns list of memories available for application specified. This command makes the platform and domain of the specified application into the active platform and domain.

```
lscript memory -supported-mem
```

Returns the section wise supported memories.

lscript section

List the sections available.

Syntax

```
lscript section [options]
```

List, add, and edit the sections available in the active domain.

Options

Option	Description
-app <application-name>	Name of application from workspace.
-name <section-name>	Name of the section to be edited.
-mem <memory-region>	Name of the memory region to be used for the section.
-size <section-size>	Size of the section.
-add	Add a new section.
-type	Type of new section to be added. Supported types are CODE, DATA, STACK, HEAP.

Returns

List of the sections with corresponding memory and size in tabular format, when no options or args are specified. Nothing, if a section successfully edited or added. Error, if the section cannot be edited or added.

Examples

```
lscript section
```

List of the sections available in the active domain along with the type, size and assigned memory.

```
lscript section -app <application-name>
```

List of the sections available for application specified. This command makes the platform and domain of the specified application into the active platform and domain.

```
lscript section -name <section-name> -mem <memory-region> -size <section-size>
```

Edit the section-name with memory and size.

```
lscript section -mem <memory-region> -size <section-size>
```

Edit all the sections with memory and size.

```
lscript section -add -name <section-name> -mem <memory-region> -size <section-size>  
-type <section-type>
```

Add a new section with section-name, memory, and size.

lscript def-mem

Returns the default memory region for the section type.

Syntax

```
lscript def-mem <memory-type>
```

Return the default memory region of the section type.

Options

Option	Description
-app <application-name>	Name of application from workspace.
-code	Return default code memory.
-data	Return default data memory.
-stack	Return default stack & heap memory.

Returns

Return the default memory region of the section type.

Examples

```
lscript def-mem -stack
```

Return default stack and heap memory-region.

```
lscript def-mem -stack -app <application-name>
```

Return default stack and heap memory region for app specified.

lscript generate

Generate a linker script.

Syntax

```
lscript generate [options]
```

Generate a linker script.

Options

Option	Description
-app <application-name>	Name of application from workspace.
-name <linkerscript name>	Name of the linker script file. The default linker script will be "newlscript.ld" if -name is not provided.
-path <path>	The directory where the linker script needs to be created, The default path will be pwd if -path is not provided.

Returns

Nothing.

Examples

```
lscript generate -name <linkerscript name> -path <path>
```

This command generates a linkerscript with the changes at the path provided. Otherwise, generate a default linker script with name "newlscript.ld".

```
lscript generate -app <application-name>
```

This command generates the default linker script for the application-name specified.

platform

Create, configure, list, and report platforms.

Syntax

```
platform <sub-command> [options]
```

Create a platform project, or perform various other operations on the platform project, based on the sub-command specified. Following sub-commands are supported.

- **active** - Set or return the active platform.
- **clean** - Clean platform.
- **config** - Configure the properties of a platform.
- **create** - Create/define a platform.
- **generate** - Build the platform.
- **list** - List all the platforms in workspace.
- **report** - Report the details of a platform.
- **read** - Read the platform settings from a file.
- **remove** - Delete the platform.
- **write** - Save the platform settings to a file.

Type "help" followed by "platform sub-command", or "platform sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command. Refer to the sub-command help for details.

Examples

Refer to the sub-command help for details.

platform active

Set/get active platform.

Syntax

```
platform active [platform-name]
```

Set or get the active platform. If a platform-name is specified, it is made the active platform. Otherwise, the name of active platform is returned. If no active platform exists, this command returns an empty string.

Options

None.

Returns

An empty string, if a platform is set as active or no active platform exists. The platform name, when the active platform is read.

Examples

```
platform active
```

Return the name of the active platform.

```
platform active zc702_platform
```

Set zc702_platform as active platform.

platform clean

Clean platform.

Syntax

```
platform clean
```

Clean the active platform in the workspace. This will clean all the components in platform such as FSBL, PMUFW, and so on.

Options

None.

Returns

Nothing. Build log will be printed on the console.

Examples

```
platform active zcu102
platform clean
```

Set zcu102 as the active platform and clean it.

platform config

Configure the active platform.

Syntax

```
platform config [options]
```

Configure the properties of active platform.

Options

Option	Description
-desc <description>	Add a brief description about the platform.
-updatehw <hw-spec>	Update the platform to use a new hardware specification file specified by <hw-spec>.
-samples <samples-dir>	Make the application template specified in <samples-dir> part of the platform. This option can only be used for acceleratable applications. "repo -apps <platform-name>" can be used to list the application templates available for the given platform-name.
-prebuilt-data <directory-name>	For expandable platforms, pre-generated hardware data specified in directory-name will be used for building user applications that do not contain accelerators. This will reduce the build time.
-make-local	Make the referenced SW components local to the platform.
-fsbl-target <processor-type>	Processor-type for which the existing fsbl has to be re-generated. This option is valid only for ZU+.
-create-boot-bsp	Generate boot components for the platform.
-remove-boot-bsp	Remove all the boot components generated during platform creation.
-fsbl-elf <fsbl.elf>	Prebuilt fsbl.elf to be used as boot component when "remove-boot-bsp" option is specified.
-pmufw-elf <pmufw.elf>	Prebuilt pmufw.elf to be used as boot component when "remove-boot-bsp" option is specified.
-extra-compiler-flags <param> <value>	Set extra compiler flag for the parameter with a provided value. Only FSBL and PMUFW are the supported parameters. If the value is not passed, the existing value will return.
-extra-linker-flags <param> <value>	Set extra linker flag for the parameter with a provided value. Only FSBL and PMUFW are the supported parameters. If the value is not passed, the existing value will return.
-reset-user-defined-flags <param>	Resets the extra compiler and linker flags. Only FSBL and PMUFW are the supported parameters.
-report <param>	Return the list of extra compiler and linker flags set to the given parameter. Only FSBL and PMUFW are the supported parameters.

Returns

Empty string, if the platform is configured successfully. Error string, if no platform is active or if the platform cannot be configured.

Examples

```
platform active zc702
platform config -desc "ZC702 with memory test application"
-samples /home/user/newDir
```

Make zc702 the active platform, configure the description of the platform, and make samples in the /home/user/newDir part of the platform.

```
platform config -updatehw /home/user/newdesign.xsa
```

Updates the platform project with the new XSA.

```
platform config -fsbl-target psu_cortexr5_0
```

Changes the FSBL target to psu_cortexr5_0.

```
platform config -extra-compiler-flags fsbl
```

Get the extra compiler flags. These are the flags added extra to the flags derived from the libraries, processor, and OS.

```
platform config -extra-compiler-flags fsbl "-DFSBL_DEBUG_INFO [platform
config
-extra-compiler-flags fsbl]"
```

Prepend -DFSBL_DEBUG_INFO to the compiler options, while building the fsbl application.

```
platform config -report fsbl
```

Return table of extra compiler and extra linker flags that are set for FSBL.

```
Platform config -create-boot-bsp
```

Create the boot components for the platform.

```
Platform config -create-boot-bsp -arch 32-bit
```

Create the boot components for the platform, creating FSBL in 32-bit. This is valid only for Zynq UltraScale+ MPSoC based platforms.

```
Platform config -remove-boot-bsp
```

Remove all the boot components generated during platform creation.

platform create

Create a new platform.

Syntax

```
platform create [options]
```

Create a new platform by importing hardware definition file. Platform can also be created from pre-defined hardware platforms. Supported pre-defined platforms are zc702, zcu102, zc706 and zed.

Options

Option	Description
-name <software-platform name>	Name of the software platform to be generated.
-desc <description>	Brief description about the software platform.
-hw <handoff-file>	Hardware description file to be used to create the platform.
-out <output-directory>	The directory where the software platform needs to be created. If the workspace is set, this option should not be used. Use of this option prevents the usage of platform in Vitis IDE.
-prebuilt	Mark the platform to be built from already built software artifacts. This option should be used only if you have existing software platform artifacts.
-proc <processor>	The processor to be used; the tool creates the default domain.
-arch <processor architecture>	32-bit or 64-bit, this is valid only for the A53 processor.
-samples <samples-directory>	Make the samples in <samples-directory>, part of the platform.
-os <os>	The OS to be used. The tool creates the default domain. This works in combination with -proc option.
-xpfm <platform-path>	Existing platform from which the projects have to be imported and made part of the current platform.
-no-boot-bsp	Mark the platform to build without generating boot components.
-arch <arch-type>	Processor architecture, <arch-type> can be 32 or 64 bits. This option is used to build the project with 32/64 bit toolchain.
-rp <slot-info>	Reconfigurable partition slot information for dfx flows. This option takes tcl dictionary with key-value pairs. Multiple slots can be passed as an array.

Returns

Empty string, if the platform is created successfully. Error string, if the platform cannot be created.

Examples

```
platform create -name "zcu102_test" -hw zcu102
```

Defines a software platform for a pre-defined hardware description file.

```
platform create -name "zcu102_test" -hw zcu102 -proc psu_cortexa53_0 -os
standalone
```

Defines a software platform for a pre-defined hardware description file. Create a default domain with standalone os running on psu_Cortexa53_0.

```
platform create -name "zcu102_32bit" -hw zcu102 -proc psu_cortexa53_0 -arch
32-bit -os standalone
```

Defines a software platform for a pre-defined hardware description file. Create a default domain with standalone os running on psu_Cortexa53_0 in 32-bit mode.

```
platform create -name "zcu102_test" -hw zcu102 -proc psu_cortexa53 -os
linux -arch 32-bit
```

Defines a software platform for a pre-defined hardware description file. Create a default domain with linux os running on psu_Cortexa53 in 32-bit.

```
platform create -xpfm /path/zc702.xpfm
```

This creates a platform project for the platform pointed by the xpfm file.

```
platform create -name "ZC702Test" -hw /path/zc702.xsa
```

Defines a software platform for a hardware description file.

```
platform create -name "testplat" -hw static.xsa -rp { id 1 hw ./hw.xsa
hw_emu ./hw_emu.xsa }
```

This creates a platform project with single slot DFX. User must specify path to hw XSA and hw_emu XSA.

```
platform create -name :testplat: -hw static.xsa -rp { { id 1 hw ./rp_1.xsa
hw_emu ./hw_emu.xsa } { id 2 hw ./rp_2.xsa hw_emu ./hw_emu.xsa } }
```

This creates a platform project with multi-slot DFX. The first slot is a default ReconfigurablePartition. For multi-slot platforms, there are multiple hw XSAs with a slot_id for each slot and potentially multiple hw_emu XSAs or single XSA.

platform generate

Build a platform.

Syntax

```
platform generate
```

Build the active platform and add it to the repository. The platform must be created through platform create command, and must be selected as active platform before building.

Options

Option	Description
-domains <domain-list>	List of domains which need to be built and added to the repository. Without this option, all the domains that are part of the platform are built.

Returns

Empty string, if the platform is generated successfully. Error string, if the platform cannot be built.

Examples

```
platform generate
```

Build the active platform and add it to repository.

```
platform generate -domains a53_standalone,r5_standalone
```

Build only a53_standalone,r5_standalone domains and add it to the repository.

platform list

List the platforms.

Syntax

List the platforms in the workspace and repository.

Options

Option	Description
-dict	List all the platforms for the workspace in Tcl dictionary format. Without this option, platforms are listed in tabular format.

Returns

List of platforms, or "No active platform present" string if no platforms exist.

Examples

```
platform list
```

Return a list of all the platforms in the workspace and repository in tabular format.

```
platform list -dict
```

Return a list of all the platforms in the workspace and repository in Tcl dictionary format.

platform report

Report the details of the active platform.

Syntax

```
platform report
```

Returns details such as domains and processors created in the active platform.

Options

None.

Returns

Table with details of active platform, or error string if no platforms exist.

Examples

```
platform report
```

Returns a table with details of the active platform.

platform read

Read from the platform file.

Syntax

```
platform read [platform-file]
```

Reads platform settings from the platform file and makes it available for edit. The platform file is created during the creation of platform itself and it contains all details of the platform such as hardware specification file, processor information, and so on.

Options

None.

Returns

Empty string, if the platform is read successfully. Error string, if the platform file cannot be read.

Examples

```
platform read <platform.spr>
```

Reads the platform from the platform.spr file.

platform remove

Delete a platform.

Syntax

```
platform remove <platform-name>
```

Delete the given platform. If the platform-name is not specified, the active platform is deleted.

Options

None.

Returns

Empty string, if the platform is deleted successfully. Error string, if the platform cannot be deleted.

Examples

```
platform remove xc702
```

Removes xc702 platform from the disk.

platform write

Write platform settings to a file.

Syntax

```
platform write
```

Writes the platform settings to platform.spr file. It can be read back using the "platform read" command.

Options

None.

Returns

Empty string, if the platform settings are written successfully. Error string, if the platform settings cannot be written.

Examples

```
platform write
```

Writes platform to platform.spr file.

domain

Create, configure, list and report domains.

Syntax

```
domain <sub-command> [options]
```

Create a domain, or perform various other operations on the domain, based on the sub-command specified. Following sub-commands are supported. active - Set/get the active domain. config - Configure the properties of a domain. create - Create a domain in the active platform. list - List all the domains in active platform. report - Report the details of a domain. remove - Delete a domain. Type "help" followed by "app sub-command", or "app sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command. Refer to the sub-command help for details.

Examples

Refer to the sub-command help for details.

domain active

Set/Get the active domain

Syntax

```
domain active [domain-name]
```

Set or get the active domain. If domain-name is specified, it is made the active domain. Otherwise, the name of the active domain is returned. If no active domain exists, this command returns an empty string.

Options

None.

Returns

Empty string, if a domain is set as active or no active domain exists. Domain name, when active domain is read.

Examples

```
domain active
```

Return the name of the active domain.

```
domain active test_domain
```

Set test_domain as active domain.

domain config

Configure the active domain.

Syntax

```
domain config [options]
```

Configure the properties of active domain.

Options

Option	Description
<code>-display-name <display name></code>	Display name of the domain.
<code>-desc <description></code>	Brief description of the domain.
<code>-sd-dir <location></code>	For domain with Linux as OS, use pre-built Linux images from this directory, while creating the PetaLinux project. This option is valid only for Linux domains.
<code>-generate-bif</code>	Generate a standard bif for the domain. Domain report shows the location of the generated bif. This option is valid only for Linux domains.

Option	Description
<code>-sw-repo <repositories-list></code>	List of repositories to be used to pick software components like drivers and libraries while generating this domain. Repository list should be a tcl list of software repository paths.
<code>-mss <mss-file></code>	Use mss from specified by <code><mss-file></code> , instead of generating mss file for the domain.
<code>-readme <file-name></code>	Add a README file for the domain, with boot instructions and so on.
<code>-inc-path <include-path></code>	Additional include path which should be added while building the application created for this domain.
<code>-lib-path <library-path></code>	Additional library search path which should be added to the linker settings of the application created for this domain.
<code>-sysroot <sysroot-dir></code>	The Linux sysroot directory that should be added to the platform. This sysroot will be consumed during application build.
<code>-boot <boot-dir></code>	Directory to generate components after Linux image build.
<code>-bif <file-name></code>	Bif file used to create boot image for Linux boot.
<code>-qemu-args <file-name></code>	File with all PS QEMU args listed. This is used to start PS QEMU.
<code>-pmuqemu-args <file-name></code>	File with all PMC QEMU args listed. This is used to start PMU QEMU.
<code>-pmcqemu-args <file-name></code>	File with all pmcqemu args listed. This is used to start pmcqemu.
<code>-qemu-data <data-dir></code>	Directory which has all the files listed in file-name provided as part of qemu-args and pmuqemu-args options.

Returns

Empty string, if the domain is configured successfully. Error string, if no domain is active or if the domain cannot be configured.

Examples

```
domain config -display-name zc702_MemoryTest
-desc "Memory test application for Zynq"
```

Configure display name and description for the active domain.

```
domain config -image "/home/user/linux_image/"
```

Create PetaLinux project from pre-built Linux image.

```
domain -inc-path /path/include/ -lib-path /path/lib/
```

Adds include and library search paths to the domain's application build settings.

domain create

Create a new domain.

Syntax

```
domain create [options]
```

Create a new domain in active platform.

Options

Option	Description
-name <domain-name>	Name of the domain.
-display-name <display_name>	The name to be displayed in the report for the domain.
-desc <description>	Brief description of the domain.
-proc <processor>	Processor core to be used for creating the domain. For SMP Linux, this can be a Tcl list of processor cores.
-arch <processor architecture>	32-bit or 64-bit. This option is valid only for A53 processors.
-os <os>	OS type. Default type is standalone.
-support-app <app-name>	Create a domain with BSP settings needed for application specified by <app-name>. This option is valid only for standalone domains. The "repo -apps" command can be used to list the available application.
-auto-generate-linux	Generate the Linux artifacts automatically.
-sd-dir <location>	For domain with Linux as OS, use pre-built Linux images from this directory, while creating the PetaLinux project. This option is valid only for Linux domains.
-sysroot <sysroot-dir>	The Linux sysroot directory that should be added to the platform. This sysroot will be consumed during application build.

Returns

Empty string, if the domain is created successfully. Error string, if the domain cannot be created.

Examples

```
domain create -name "ZUdomain" -os standalone -proc psu_cortexa53_0  
-support-app {Hello World}
```

Create a standalone domain and configure settings needed for a Hello World template application.

```
domain create -name "SMPLinux" -os linux  
-proc {ps7_cortexa9_0 ps7_cortexa9_1}
```

Create a Linux domain named SMPLinux for processor cores ps7_cortexa9_0 ps7_cortexa9_1 in the active platform.

```
domain create -name a53_0_Standalone -os standalone  
-proc psu_cortexa53_0 -arch 32-bit
```

Create a standalone domain for a53_0 processor for 32-bit mode.

domain list

List domains.

Syntax

```
domain list
```

List domains in the active platform.

Options

Option	Description
-dict	List all the domains for the active platform in Tcl dictionary format. Without this option, domains are listed in tabular format.

Returns

List of domains in the active platform, or empty string if no domains exist.

Examples

```
platform active
```

platform1

```
domain list
```

Display all the domain created in platform1 in tabular format.

```
domain list -dict
```

Display all the domain created in platform1 in Tcl dictionary format.

domain remove

Delete a domain.

Syntax

```
domain remove [domain-name]
```

Delete a domain from active platform. If a domain-name is not specified, the active domain is deleted.

Options

None.

Returns

Empty string, if the domain is deleted successfully. Error string, if the domain deletion fails.

Examples

```
domain remove test_domain
```

Removes test_domain from the active platform.

domain report

Report the details of a domain.

Syntax

```
domain report [domain-name]
```

Return details such as platform, processor core, OS, and so on. of a domain. If domain-name is not specified, details of the active domain are reported.

Options

None.

Returns

Table with details of a domain, if domain-name or active domain exists. Error string, if active domain does not exist and domain-name is not specified.

Examples

```
domain report
```

Return a table with details of the active domain.

bsp

Configure BSP settings of a baremetal domain.

Syntax

```
bsp <sub-command> [options]
```

Configure the BSP settings, including the library, driver, and OS version of a active domain, based on the sub-command specified. Following sub-commands are supported. config - Modify the configurable parameters of BSP settings. getdrivers - List IP instance and its driver. getlibs - List the libraries from BSP settings. getos - List os details from BSP settings. listparams - List the configurable parameters of os/proc/library. regenerate - Regenerate BSP sources. reload - Revert the BSP settings to the earlier saved state. write - Save the BSP edits. removelib - Remove library from bsp settings. setdriver - Sets the driver for the given IP instance. setlib - Sets the given library. setosversion - Sets version for the given OS. Type "help" followed by "bsp sub-command", or "bsp sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command. Refer to the sub-command help for details.

Examples

Refer to the sub-command help for details.

bsp config

Configure parameters of BSP settings.

Syntax

```
bsp config <param> <value>
```

Set/get/append value to the configurable parameters. If <param> is specified and <value> is not specified, return the value of the parameter. If <param> and <value> are specified, set the value of parameter. Use "bsp list-params <-os/-proc/-driver>" to know configurable parameters of OS/processor/driver.

Options

Option	Description
-append <param> <value>	Append the given value to the parameter.

Returns

Nothing, if the parameter is set/appended successfully. Current value of the parameter if <value> is not specified. Error string, if the parameter cannot be set/appended.

Examples

```
bsp config -append extra_compiler_flags "-pg"
```

Append -pg to extra_compiler_flags.

```
bsp config stdin
```

Return the current value of stdin.

```
bsp config stdin ps7_uart_1
```

Set stdin to ps7_uart_1 .

bsp getdrivers

List drivers.

Syntax

```
bsp getdrivers
```

Return the list of drivers assigned to IP in BSP.

Options

Option	Description
-dict	Return the result as <IP-name driver:version> pairs.

Returns

Table with IP, its corresponding driver, and driver version. Empty string, if there are no IPs.

Examples

```
bsp getdrivers
```

Return the list of IPs and their drivers.

bsp getlibs

List libraries added in the BSP settings.

Syntax

```
bsp getlibs
```

Display list of libraries added in the BSP settings.

Options

Option	Description
-dict	Return the result as <lib-name version> pairs.

Returns

List of libraries. Empty string, if no libraries are added.

Examples

```
bsp getlibs
```

Return the list of libraries added in bsp settings of active domain.

bsp getos

Display OS details from BSP settings.

Syntax

```
bsp getos
```

Displays the current OS and version.

Options

Option	Description
-dict	Return the result as <os-name version> pair.

Returns

OS name and version.

Examples

```
bsp getos
```

Return OS name and version from the BSP settings of the active domain.

bsp listparams

List the configurable parameters of the BSP.

Syntax

```
bsp listparams <option>
```

List the configurable parameters of the <option>.

Options

Option	Description
-lib <lib-name>	Return the configurable parameters of the library in BSP.
-os	Return the configurable parameters of the OS in BSP.
-proc	Return the configurable parameters of the processor in BSP.

Returns

Parameter names. Empty string, if no parameters exist.

Examples

```
bsp listparams -os
```

List all the configurable parameters of OS in the BSP settings.

bsp regenerate

Regenerate BSP sources.

Syntax

```
bsp regenerate
```

Regenerate the sources with the modifications made to the BSP.

Options

None.

Returns

Nothing, if the BSP is generated successfully. Error string, if the BSP cannot be generated.

Examples

```
bsp regenerate
```

Regenerate the BSP sources with the changes to the BSP settings applied.

bsp removelib

Remove library from BSP settings.

Syntax

```
bsp removelib -name <lib-name>
```

Remove the library from BSP settings of the active domain. The library settings will come into effect only when the platform is generated. Settings can also be saved by running 'bsp write' if the user wishes to exit xsct without generating platform and revisit later.

Options

Option	Description
-name <lib-name>	Library to be removed from BSP settings. This is the default option, so lib-name can be directly specified as an argument without using this option.

Returns

Nothing, if the library is removed successfully. Error string, if the library cannot be removed.

Examples

```
bsp removelib -name xilffs
```

Remove xilffs library from BSP settings.

bsp setdriver

Set the driver to IP.

Syntax

```
bsp setdriver [options]
```

Set specified driver to the IP core in BSP settings of active domain.

Options

Option	Description
-driver <driver-name>	Driver to be assigned to an IP.
-ip <ip-name>	IP instance for which the driver has to be added.
-ver <version>	Driver version.

Returns

Nothing, if the driver is set successfully. Error string, if the driver cannot be set.

Examples

```
bsp setdriver -ip ps7_uart_1 -driver generic -ver 2.0
```

Set the generic driver for the ps7_uart_1 IP instance for the BSP.

bsp setlib

Adds the library to the BSP settings.

Syntax

```
bsp setlib [options]
```

Queues the library for addition to the active BSP. 'bsp write' will commit the queued libraries to the mss. The newly added libraries become available to the application projects after the platform is generated. If the user wants to build the platform from GUI without committing the queued libraries to mss, then the project must be cleaned first.

Options

Option	Description
-name <lib-name>	Library to be added to the BSP settings. This is the default option, so lib-name can be directly specified as an argument without using this option.
-ver <version>	Library version.

Returns

Nothing, if the library is set successfully. Error string, if the library cannot be set.

Examples

```
bsp setlib -name xilffs
```

Add the xilffs library to the BSP settings.

bsp setosversion

Set the OS version.

Syntax

```
bsp setosversion [options]
```

Set OS version in the BSP settings of the active domain. Latest version is added by default.

Options

Option	Description
-ver <version>	OS version.

Returns

Nothing, if the OS version is set successfully. Error string, if the OS version cannot be set.

Examples

```
bsp setosversion -ver 6.6
```

Set the OS version 6.6 in the BSP settings of the active domain.

library

Library project management

Syntax

```
library <sub-command> [options]
```

Create a library project, or perform various other operations on the library project, based on the sub-command specified. Following sub-commands are supported. build - Build the library project. clean - Clean the library project. config - Configure C/C++ build settings of the library project. create - Create a library project. list - List all the library projects in workspace. remove - Delete the library project. report - Report the details of the library project. Type "help" followed by "library sub-command", or "library sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command.

Examples

See sub-command help for examples.

library build

Build library project.

Syntax

```
library build -name <project-name>
```

Build the library project specified by <project-name> in the workspace. "-name" switch is optional, so <project-name> can be specified directly, without using -name.

Options

Option	Description
-name <project-name>	Name of the library project to be built.

Returns

Nothing, if the library project is built successfully. Error string, if the library project build fails.

Examples

```
library build -name lib1
```

Build lib1 library project.

library clean

Clean library project.

Syntax

```
library clean -name <project-name>
```

Clean the library project specified by <project-name> in the workspace. "-name" switch is optional, so <project-name> can be specified directly, without using -name.

Options

Option	Description
-name <project-name>	Name of the library project to be clean built.

Returns

Nothing, if the library project is cleaned successfully. Error string, if the library project build clean fails.

Examples

```
library clean -name lib1
```

Clean lib1 library project.

library create

Create a library project.

Syntax

```
library create -name <project-name> -type <library-type> -platform  
<platform>
```

-domain <domain> -sysproj <system-project> Create a library project using an existing platform, and domain. If <platform>, <domain>, and <sys-config> are not specified, the active platform and domain are used for creating the library project. For creating a library project and adding it to an existing system project, refer to the next use case.

```
library create -name <project-name> -type <library-type> -sysproj <system-  
project>
```

-domain <domain> Create a library project for domain specified by <domain> and add it to system project specified by <system-project>. If <system-project> exists, platform corresponding to this system project are used for creating the library project. If <domain> is not specified, the active domain is used.

Options

Option	Description
<code>-name <project-name></code>	Project name that should be created.
<code>-type <library-type></code>	<library-type> can be 'static' or 'shared'.
<code>-platform <platform-name></code>	Name of the platform. Use "repo -platforms" to list available pre-defined platforms.
<code>-domain <domain-name></code>	Name of the domain. Use "platform report <platform-name>" to list the available domains in a platform.
<code>-sysproj <system-project></code>	Name of the system project. Use "sysproj list" to know the available system projects in the workspace.

Returns

Nothing, if the library project is created successfully. Error string, if the library project creation fails.

Examples

```
library create -name lib1 -type static -platform zcu102 -domain  
a53_standalone
```

Create a static library project with name 'lib1', for the platform zcu102, which has a domain named a53_standalone domain.

```
library create -name lib2 -type shared -sysproj test_system -domain  
test_domain
```

Create shared library project with name 'lib2' and add it to system project test_system.

library list

List library projects.

Syntax

List all library projects in the workspace.

Options

None.

Returns

List of library projects in the workspace. If no library projects exist, an empty string is returned.

Examples

```
library list
```

Lists all the library projects in the workspace.

library remove

Delete library project.

Syntax

```
library remove [options] <project-name>
```

Delete a library project from the workspace.

Options

None.

Returns

Nothing, if the library project is deleted successfully. Error string, if the library project deletion fails.

Examples

```
library remove lib1
```

Removes lib1 from workspace.

library report

Report details of the library project.

Syntax

```
library report <project-name>
```

Return details such as the platform, domain, and so on of the library project.

Options

None.

Returns

Details of the library project, or error string, if library project does not exist.

Examples

```
app report lib1
```

Return all the details of library lib1.

checkvalidrmxsa

Check if RM XSA is suitable for static XSA.

Syntax

```
checkvalidrmxsa -hw <static hw spec file> -rm-hw <rm hw spec file>
```

To check if the RM XSA is suitable to work with the static hardware XSA.

Options

None.

Returns

If successful, returns true if the RM hardware XSA is a fit for the static hardware XSA. Returns false if not. Otherwise, it returns an error.

Examples

```
checkvalidermxsa -hw static.xsa -rm-hw rm.xsa
```

Returns true if RM XSA can be used along with the static XSA.

isstaticxsa

Check if hardware design is a static XSA.

Syntax

```
isstaticxsa <hw spec file>
```

Checks if the hardware design is a static XSA.

Options

None.

Returns

If successful, returns true if hardware design is static, returns false if hardware design is not static. Otherwise, it returns an error.

Examples

```
isstaticxsa static.xsa
```

Returns true if XSA is static.

ishwexpandable

Check if hardware design is expandable.

Syntax

```
ishwexpandable <hw spec file>
```

Checks if the hardware design is expandable or fixed.

Options

None.

Returns

If successful, returns true if hardware design is expandable/extensible, returns false if hardware design is fixed. Otherwise, it returns an error.

Examples

```
ishwexpandable system.xsa
```

Returns true if XSA is expandable/extensible.

createdts

Creates device tree.

Syntax

```
createdts [options]
```

Create a device tree for the hardware definition file.

Options

Option	Description
-platform-name <software-platform name>	Name of the software platform to be generated.
-board <board name>	Board name for device tree to be generated. Board names available at <DTG Repo>/device_tree/data/kernel_dtsi.
-hw <handoff-file>	Hardware description file to be used to create the device tree.
-out <output-directory>	The directory where the software platform needs to be created. Workspace will be default directory, if this option is not specified.
-local-repo <directory location>	Location of the directory where bsp for git repo is available. Device tree repo will be cloned from git, if this option is not specified.
-git-url <Git URL>	Git URL of the dtg repo to be cloned. Default repo is https://github.com/Xilinx/device-tree-xlnx.git.
-git-branch <Git Branch>	Git branch to be checked out. 'xlnx_rel_v<Vitis-release>' is selected by default.
-zocl	Set zocl flag to enable zocl driver support, default set to False. zocl should only be used when the designs are PL enabled. Only master and xlnx_rel_v2021.2 branch supports zocl property.

Option	Description
-overlay	Set overlay flag to enable device-tree overlay support, default set to False.
-dtsi <custom-dtsi-file list>	Include custom-dtsi file in the device tree, if specified. The filepaths must be in the list format.
-compile	Specify this option to compile the generated dts to create dtb. If this option is not specified, users can manually use dts to compile dtb. For example, dtc -I dts -O dtb -o <file_name>.dtb <file_name>.dts Compile dts(device tree source) or dtsi(device tree source include) files. dtc -I dts -O dtb -f <file_name>.dts -o <file_name>.dtb Convert dts(device tree source) to dtb(device tree blob). dtc -I dtb -O dtb -f <file_name>.dtb -o <file_name>.dts Convert dtb(device tree blob) to dts(device tree source).
-update	Set update flag to enable existing device tree platform to update with new xsa.

Note(s)

- This command is a shortcut for creating a device tree domain and generating the device tree. It clones the device tree repo, creates a platform with device_tree as OS, and configures and generates the platform to create dts. -zocl should only be used when the designs are PL enabled. Only master and xlnx_rel_v2021.2 branch supports zocl property. Git 1.5.4 or later is required to avoid any issues with the git commands used by the createdts command.

Returns

None.

Examples

```
createdts -hw zcu102.xsa -platform-name my_devicetree
```

Create a device tree for the handoff-file with default repo as "https://github.com/Xilinx/device-tree-xlnx.git" and default branch as "xlnx_rel_v<Vitis-release>".

```
createdts -hw zcu102.xsa -platform-name my_devicetree -git-url <Git URL>
          -git-branch <Git Branch>
```

Create a device tree for the handoff-file with user repo as repo mentioned in <Git URL> and user branch as <Git Branch>.

```
createdts -hw zc702.xsa -platform-name my_devicetree
          -local-repo /my_local_git_repo
```

Create a device tree for the handoff-file and use the local repo.

```
createdts -hw vck190.xsa -platform-name my_devicetree
          -out /device-tree_output_directory
```

Create a device tree at the out directory specified by device-tree output directory.

```
createdts -hw zcu102.xsa -platform-name my_devicetree -overlay  
-zocl -compile
```

Create device tree for the handoff-file with overlay and zocl node. Compile flag compiles the device tree blob file from the DTS.

```
createdts -hw zcu102.xsa -platform-name my_devicetree -board <Board Name>
```

Creates a device tree adding board value to the library, Board names available at <DTG Repo>/device_tree/data/kernel_dtsi.

```
createdts -update -hw newdesign.xsa
```

Updates existing device tree platform with new XSA.

```
createdts -hw vck190 -platform-name vck190 -out <out_dir>  
-dtsi [list path/system-conf.dtsi path/system-user.dtsi]
```

Create device tree with custom-dtsi-files included.

setws

Set Vitis workspace

Syntax

```
setws [OPTIONS] [path]
```

Set Vitis workspace to <path>, for creating projects. If <path> does not exist, then the directory is created. If <path> is not specified, then current directory is used.

Options

Option	Description
-switch <path>	Close existing workspace and switch to new workspace.

Returns

Nothing if the workspace is set successfully. Error string, if the path specified is a file.

Examples

```
setws /tmp/wrk/wksp1
```

Set the current workspace to /tmp/wrk/wksp1.

```
setws -switch /tmp/wrk/wksp2
```

Close the current workspace and switch to new workspace /tmp/wrk/wksp2.

getws

Get Vitis workspace.

Syntax

```
getws
```

Return the current vitis workspace.

Returns

Current workspace.

app

Application project management.

Syntax

```
app <sub-command> [options]
```

Create an application project, or perform various other operations on the application project, based on <sub-command> specified. Following sub-commands are supported. build - Build the application project. clean - Clean the application project. config - Configure C/C++ build settings of the application project. create - Create an application project. list - List all the application projects in workspace. remove - Delete the application project. report - Report the details of the application project. switch - Switch application project to refer another platform. Type "help" followed by "app sub-command", or "app sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command. Refer to the sub-command help for details.

Examples

Refer to the sub-command help for examples.

app build

Build application.

Syntax

```
app build -name <app-name>
```

Build the application specified by <app-name> in the workspace. "-name" switch is optional, so <app-name> can be specified directly, without using -name.

Options

Option	Description
-name <app-name>	Name of the application to be built.
-all	Option to Build all the application projects.

Returns

Nothing. Build log will be printed on the console.

Examples

```
app build -name helloworld
```

Build Hello World application.

```
app build -all
```

Build all the application projects in the workspace.

app clean

Clean application.

Syntax

```
app clean -name <app-name>
```

Clean the application specified by <app-name> in the workspace. "-name" switch is optional, so <app-name> can be specified directly, without using -name.

Options

Option	Description
-name <app-name>	Name of the application to be clean built.

Returns

Nothing. Build log will be printed on the console.

Examples

```
app clean -name helloworld
```

Clean Hello World application.

app config

Configure C/C++ build settings of the application.

Syntax

Configure C/C++ build settings for the specified application. Following settings can be configured for applications: assembler-flags : Miscellaneous flags for assembler build-config : Get/set build configuration compiler-misc : Compiler miscellaneous flags compiler-optimization : Optimization level define-compiler-symbols : Define symbols. Ex. MYSYMBOL include-path : Include path for header files libraries : Libraries to be added while linking library-search-path : Search path for the libraries added linker-misc : Linker miscellaneous flags linker-script : Linker script for linking undef-compiler-symbols : Undefine symbols. Ex. MYSYMBOL

```
app config -name <app-name> <param-name>
```

Get the value of configuration parameter <param-name> for the application specified by <app-name>.

```
app config [OPTIONS] -name <app-name> <param-name> <value>
```

Set/modify/remove the value of configuration parameter <param-name> for the application specified by <app-name>.

Options

Option	Description
-name	Name of the application.
-set	Set the configuration parameter value to new <value>.
-get	Get the configuration parameter value.

Option	Description
-add	Append the new <value> to configuration parameter value. Add option is not supported for compiler-optimization
-info	Displays more information like possible values and possible operations about the configuration parameter. A parameter name must be specified when this option is used.
-remove	Remove <value> from the configuration parameter value. Remove option is not supported for assembler-flags, build-config, compiler-misc, compiler-optimization, linker-misc, and linker-script.

Returns

Depends on the arguments specified. <none> List of parameters available for configuration and description of each parameter.

<parameter name>: Parameter value, or error, if unsupported parameter is specified.

<parameter name> <parameter value>: Nothing if the value is set successfully, or error, if unsupported parameter is specified.

Examples

```
app config -name test build-config
```

Return the current build configuration for the application named test.

```
app config -name test define-compiler-symbols FSBL_DEBUG_INFO
```

Add -DFSBL_DEBUG_INFO to the compiler options, while building the test application.

```
app config -name test -remove define-compiler-symbols FSBL_DEBUG_INFO
```

Remove -DFSBL_DEBUG_INFO from the compiler options, while building the test application.

```
app config -name test -set compiler-misc { -c -fmessage-length=0 -MT "$@" }
```

Set {-c -fmessage-length=0 -MT "\$@"} as compiler miscellaneous flags for the test application.

app config -name test -append compiler-misc {-pg} Add {-pg} to compiler miscellaneous flags for the test application.

```
app config -name test -info compiler-optimization
```

Changing the compiler-optimization in the compiler options of an application.

```
app config -name test -set compiler-optimization {Optimize for size (-Os)}
```

Display more information about possible values and default values for compiler optimization level.

app create

Create an application.

Syntax

```
app create [options] -platform <platform> -domain <domain>
```

-sysproj <system-project> Create an application using an existing platform and domain, and add it to a system project. If <platform> and <domain> are not specified, then active platform and domain are used for creating the application. If <system-project> is not specified, then a system project is created with name appname_system. For creating applications and adding them to existing system project, refer to next use case. Supported options are: -name, -template.

```
app create [options] -sysproj <system-project> -domain <domain>
```

Create an application for domain specified by <domain> and add it to system project specified by <system-project>. If <system-project> exists, platform corresponding to this system project are used for creating the application. If <domain> is not specified, then active domain is used. Supported options are: -name, -template.

```
app create [options] -hw <hw-spec> -proc <proc-instance>
```

Create an application for processor core specified <proc-instance> in HW platform specified by <hw-spec>. Supported options are: -name, -template, -os, -lang, -arch.

Options

Option	Description
<code>-name <application-name></code>	Name of the application to be created.
<code>-platform <platform-name></code>	Name of the platform. Use "repo -platforms" to list available pre-defined platforms.
<code>-domain <domain-name></code>	Name of the domain. Use "platform report <platform-name>" to list the available system configurations in a platform.
<code>-hw <hw-spec></code>	HW specification file exported from Vivado (XSA).
<code>-sysproj <system-project></code>	Name of the system project. Use "sysproj list" to know available system projects in the workspace.
<code>-proc <processor></code>	Processor core for which the application should be created.
<code>-template <application template></code>	Name of the template application. Default is "Hello World". Use "repo -apps" to list available template applications.
<code>-os <os-name></code>	OS type. Default type is standalone.
<code>-lang <programming language></code>	Programming language can be c or c++.
<code>-arch <arch-type></code>	Processor architecture, <arch-type> can be 32 or 64 bits. This option is used to build the project with 32/64 bit toolchain.

Returns

Nothing, if the application is created successfully. Error string, if the application creation fails.

Examples

```
app create -name test -platform zcu102 -domain a53_standalone
```

Create Hello World application named test, for the platform zcu102, with a domain named a53_standalone.

```
app create -name zqfsbl -hw zc702 -proc ps7_cortexa9_0 -os standalone  
-template "Zynq FSBL"
```

Create Zynq FSBL application named zqfsbl for ps7_cortexa9_0 processor core, in zc702 HW platform.

```
app create -name memtest -hw /path/zc702.xsa -proc ps7_cortexa9_0 -os  
standalone  
-template "Memory Tests"
```

Create Memory Test application named memtest for ps7_cortexa9_0 processor core, in zc702.xsa HW platform.

```
app create -name test -sysproj test_system -domain test_domain
```

Create Hello World application project with name test and add it to system project test_system.

app list

List applications.

Syntax

```
app list
```

List all applications for in the workspace.

Options

Option	Description
-dict	List all the applications for the workspace in Tcl dictionary format. Without this option, applications are listed in tabular format.

Returns

List of applications in the workspace. If no applications exist, "No application exist" string is returned.

Examples

```
app list
```

Lists all the applications in the workspace in tabular format.

```
app list -dict
```

Lists all the applications in the workspace in Tcl dictionary format.

app remove

Delete application.

Syntax

```
app remove <app-name>
```

Delete an application from the workspace.

Options

None.

Returns

Nothing, if the application is deleted successfully. Error string, if the application deletion fails.

Examples

```
app remove zynqapp
```

Removes zynqapp from workspace.

app report

Report details of the application.

Syntax

```
app report <app-name>
```

Return details such as the platform, domain, processor core, and OS of an application.

Options

None.

Returns

Details of the application, or error string, if application does not exist.

Examples

```
app report test
```

Return all the details of application test.

app switch

Switch the application to use another domain/platform.

Syntax

```
app switch -name <app-name> -platform <platform-name> -domain <domain-name>
```

Switch the application to use another platform and domain. If the domain name is not specified, application will be moved to the first domain which is created for the same processor as current domain. This option is supported if there is only one application under this platform.

```
app switch -name <app-name> -domain <domain-name>
```

Switch the application to use another domain within the same platform. New domain should be created for the same processor as current domain.

Options

Option	Description
-name <application-name>	Name of the application to be switched.
-platform <platform-name>	Name of the new Platform. Use "platform -list" to list the available platforms.
-domain <domain-name>	Name of the new domain. Use "domain -list" to list available domain in the active platform.

Returns

Nothing if application is switched successfully, or error string, if given platform project does not exist or given platform project does not have valid domain.

Examples

```
app switch -name helloworld -platform zcu102
```

Switch the Hello World application to use zcu102 platform.

sysproj

System project management.

Syntax

```
sysproj <sub-command> [options]
```

Build, list and report system project, based on <sub-command> specified. Following sub-commands are supported. build - Build the system project. clean - Clean the system project. list - List all system projects in workspace. remove - Delete the system project. report - Report the details of the system project. Type "help" followed by "sysproj sub-command", or "sysproj sub-command" followed by "-help" for more details.

Options

None.

Returns

Depends on the sub-command.

Examples

See sub-command help for examples.

sysproj build

Build system project.

Syntax

```
sysproj build -name <sysproj-name>
```

Build the application specified by <sysproj-name> in the workspace. "-name" switch is optional, so <sysproj-name> can be specified directly, without using -name.

Options

Option	Description
-name <sysproj-name>	Name of the system project to be built.
-all	Option to build all the system projects.

Examples

```
sysproj build -name helloworld_system
```

Build the system project specified.

```
sysproj build -all
```

Build all the system projects in the workspace.

sysproj clean

Clean application.

Syntax

```
sysproj clean -name <app-name>
```

Clean the application specified by <sysproj-name> in the workspace. "-name" switch is optional, so <sysproj-name> can be specified directly, without using -name.

Options

Option	Description
-name <sysproj-name>	Name of the application to be clean built.

Returns

Nothing, if the application is cleaned successfully. Error string, if the application build clean fails.

Examples

```
sysproj clean -name helloworld_system
```

Clean-build the system project specified.

sysproj list

List system projects.

Syntax

```
sysproj list
```

List all system projects in the workspace.

Options

None.

Returns

List of system projects in the workspace. If no system project exist, an empty string is returned.

Examples

```
sysproj list
```

List all system projects in the workspace.

sysproj remove

Delete system project.

Syntax

```
sysproj remove [options]
```

Delete a system project from the workspace.

Options

None.

Returns

Nothing, if the system project is deleted successfully. Error string, if the system project deletion fails.

Examples

```
sysproj remove test_system
```

Delete test_system from workspace.

sysproj report

Report details of the system project.

Syntax

```
sysproj report <sysproj-name>
```

Return the details such as the platform and domain of a system project.

Options

None.

Returns

Details of the system project, or error string, if system project does not exist.

Examples

```
sysproj report test_system
```

Return all the details of the system project test_system.

importprojects

Import projects to workspace.

Syntax

```
importprojects <path>
```

Import all the Vitis projects from <path> to workspace.

Returns

Nothing, if the projects are imported successfully. Error string, if project path is not specified or if the projects cannot be imported.

Examples

```
importprojects /tmp/wrk/wksp1/hello1
```

Import Vitis project(s) into the current workspace.

importsources

Import sources to an application project.

Syntax

```
importsources [OPTIONS]
```

Import sources from a path to application project in workspace.

Options

Option	Description
-name <project-name>	Application Project to which the sources should be imported.

Option	Description
<code>-path <source-path></code>	Path from which the source files should be imported. If <code><source-path></code> is a file, it is imported to application project. If <code><source-path></code> is a directory, all the files/sub-directories from the <code><source-path></code> are imported to application project. All existing source files will be overwritten in the application, and new files will be copied. Linker script will not be copied to the application directory, unless <code>-linker-script</code> option is used.
<code>-soft-link</code>	Links the sources from source-path and does not copy the source.
<code>-target-path <dir-path></code>	Directory to which the sources have to be linked or copied. If <code>target-path</code> option is not used, source files will be linked or copied to "src" directory.
<code>-linker-script</code>	Copies the linker script as well.

Returns

Nothing, if the project sources are imported successfully. Error string, if invalid options are used or if the project sources cannot be read/imported.

Examples

```
importsources -name hello1 -path /tmp/wrk/wksp2/hello2
```

Import the 'hello2' project sources to 'hello1' application project without the linker script.

```
importsources -name hello1 -path /tmp/wrk/wksp2/hello2 -linker-script
```

Import the 'hello2' project sources to 'hello1' application project along with the linker script.

```
importsources -name hello1 -path /tmp/wrk/wksp2/hello_app -soft-link
```

Create a soft-link to hello1 application project from hello_app application project.

toolchain

Set or get toolchain used for building projects.

Syntax

```
toolchain
```

Return a list of available toolchains and supported processor types.

```
toolchain <processor-type>
```

Get the current toolchain for `<processor-type>`.

```
toolchain <processor-type> <tool-chain>
```

Set the <toolchain> for <processor-type>. Any new projects created will use the new toolchain during build.

Returns

Depends on the arguments specified.

<none>: List of available toolchains and supported processor types.

<processor-type>: Current toolchain for processor-type.

<processor-type><tool-chain>: Nothing if the tool-chain is set, or error, if unsupported tool-chain is specified.

XSCT Use Cases

XSCT can be used in various scenarios in the development, debugging, verification, and deployment cycles. XSCT inherits high-level, interpreted, and dynamic programming features from Tcl, which makes the programming simple and powerful.

XSCT can inter-operate the workspace together with the AMD Vitis™ IDE. When creating and managing projects, XSCT launches the Vitis IDE in the background. XSCT workspaces can be seamlessly used with the Vitis IDE and vice versa. When you are working in the Vitis IDE, equivalent XSCT commands will be printed in the console in most use cases. This can help you create scripts for batching and automation when actions need to be executed repeatedly.

Note: At any point in time, a workspace can either be used only from Vitis IDE or XSCT.

Common Use Cases

- **Checking the JTAG status of the board:** In the new board bring-up phase, after verifying the power circuits, the first job for hardware verification is to test the JTAG status; checking whether the FPGA or SoC device can be scanned, and whether the processors can be found properly. XSCT can do this job with JTAG access and target connection management commands such as 'jtag targets', 'connect', and 'targets'. If you suspect that the board is in an abnormal status and you need to check the basic hardware, it is also recommended to check the JTAG and processor status.
- **Initializing the board with a single script through JTAG:** In some debugging cases (for example, debugging a PL module that needs a PS generated clock), the PS simply needs to be initialized into a certain status. Running customized initialization scripts can be faster and more lightweight than launching runs with the Vitis IDE. The Vitis IDE shows the equivalent XSCT debug commands in the console. To repeat an initialization cycle easily, copy these commands into a Tcl file and use XSCT to execute this Tcl script.
- **Loading U-Boot with a single script through JTAG:** If you need to customize U-Boot, the easiest way to test and iterate is to use XSCT to initialize the board, load the U-Boot binary into DDR, and run it. This can be executed on the fly. Otherwise, you might have to package the `boot.bin` file and write it to an SD card or the flash memory every time you update the code.

- **Reading and writing registers with or without applications:** When debugging peripherals or their drivers, the status of the peripheral registers is important. The status can be read from XSCT or it can be viewed in the Vitis IDE memory view. Using XSCT commands to read and write registers is quick and lightweight. The register read and write commands can be written into a script to automate repeated processes. You can also save the register values into a file for comparison.

Changing Compiler Options of an Application Project

An example XSCT session that demonstrates creating an empty application for Cortex®-A53 processor, by adding the compiler option `-std=c99` is as follows.

```
setws /tmp/wrk/workspace
app create -name test_a53 -hw /tmp/wrk/system.xsa -os standalone -proc
psu_cortexa53_0 -template {Empty Application(C)}
importsources -name test_a53 -path /tmp/sources/
app config -name test_a53 -add compiler-misc {-std=c99}
app build -name test_a53
```

Creating an Application Project Using an Application Template (Zynq UltraScale+ MPSoC FSBL)

The following is an example XSCT session that demonstrates creating a FSBL project for a Cortex-A53 processor.

Note: Creating an application project creates a BSP project by adding the necessary libraries and setting compiler options automatically. `FSBL_DEBUG_DETAILED` symbol is added to FSBL for debug messages.

```
setws /tmp/wrk/workspace
app create -name a53_fsbl -hw /tmp/wrk/system.xsa -os standalone -proc
psu_cortexa53_0 -template {Zynq MP FSBL}
app config -name a53_fsbl define-compiler-symbols {FSBL_DEBUG_INFO}
app build -name a53_fsbl
```

Creating an FSBL Application Project Using Manually Created Domain (Zynq UltraScale+ MPSoC FSBL)

The following is an example XSCT session that demonstrates creating a FSBL project for a Cortex-A53 processor by manually creating platform, domain and application. Configuration option `zynqmp_fsbl_bsp` is set for FSBL compiler optimization options.

```
setws /tmp/wrk/workspace
platform create -name HW1 -hw zcu102 -no-boot-bsp
domain create -name A53_Standalone -os standalone -proc psu_cortexa53_0
domain active A53_Standalone
bsp setlib -name xilffs
bsp setlib -name xilsecure
bsp setlib -name xilpm
bsp config zynqmp_fsbl_bsp true

platform generate
app create -name a53_fsbl -platform HW1 -template "Zynq MP FSBL" -domain
A53_Standalone -lang c
app build -name a53_fsbl
```

Creating a Bootable Image and Program the Flash

An example XSCT session that demonstrates the creation of a "Hello World" application is shown in the following snippet. It also shows the creation of a bootable image using the applications along with bitstream by building the system project and programming the image onto the flash.

Note: The Vitis environment creates a platform project and system project when an application project is created. The platform project includes boot components such as FSBL, which are required for initializing a device. This example assumes that you are using the ZC702 board, and uses `-flash_type qspi_single` as an option with `program_flash`.

```
setws /tmp/wrk/workspace
app create -name a9_fsbl -hw /tmp/wrk/system.xsa -os standalone -proc
ps7_cortexa9_0 -template {Hello World}
app build -name a9_hello
# Build the system project. This builds the platform project to generate
fsbl.elf
# and creates a bif file and runs Bootgen to create a boot image (BOOT.BIN)
sysproj build -name a9_hello_system
# Modify the bif and run Bootgen if needed
# exec bootgen -arch zynq -image output.bif -w -o /tmp/wrk/BOOT.bin
# Program the flash and verify the flash device
exec program_flash -f /tmp/wrk/BOOT.bin -flash_type qspi_single -
blank_check -verify -cable type xilinx_tcf url tcp:localhost:3121
```

Debugging a Program Already Running on the Target

Xilinx System Debugger Command-line Interface (XSDB) can be used to debug a program which is already running on the target (for example, booting from flash). Connect to the target and set the symbol file for the program running on the target. This method can also be used to debug Linux kernel booting from flash. For best results, the code running on the target should be compiled with the debug information.

The following is an example of debugging a program already running on the target. For demo purpose, the program has been stopped at `main()`, before this example session.

```
# Connect to the hw_server

xsdb% conn -url TCP:xhdbfarmc7:3121
tcfchan#0
xsdb% Info: Arm Cortex-A9 MPCore #0 (target 2) Stopped at 0x1005a4
(Hardware Breakpoint)
xsdb% Info: Arm Cortex-A9 MPCore #1 (target 3) Stopped at 0xfffffe18
(Suspended)

# Select the target on which the program is running and specify the symbol
file using the
# memmap command

xsdb% targets 2
xsdb% memmap -file dhystone/Debug/dhystone.elf

# When the symbol file is specified, the debugger maps the code on the
target to the symbol
# file. bt command can be used to see the back trace. Further debug is
possible, as shown in
# the first example

xsdb% bt
  0 0x1005a4 main(): ../src/dhry_1.c, line 79
  1 0x1022d8 _start()+88
  2 unknown-pc
```

Debugging Applications on Zynq UltraScale+ MPSoC

Note: For simplicity, this help page assumes that AMD Zynq™ UltraScale+™ MPSoC boots up in JTAG bootmode. The flow described here can be applied to other boot modes too, with minor changes.

When Zynq UltraScale+ MPSoC boots up JTAG bootmode, all the Cortex®-A53 and Cortex®-R5F cores are held in reset. Users must clear resets on each core, before debugging on these cores.

The `rst` command in XSCT can be used to clear the resets. `rst -processor` clears reset on an individual processor core. `rst -cores` clears resets on all the processor cores in the group (APU or RPU), of which the current target is a child. For example, when Cortex-A53 #0 is the current target, `rst -cores` clears resets on all the Cortex-A53 cores in APU.

Below is an example XSCT session that demonstrates standalone application debug on Cortex-A53 #0 core on Zynq UltraScale+ MPSoC.

Note: Similar steps can be used for debugging applications on Cortex-R5F cores and also on Cortex-A53 cores in 32 bit mode. However, the Cortex-A53 cores must be put in 32 bit mode, before debugging the applications. This should be done after POR and before the Cortex-A53 resets are cleared.

```
#connect to remote hw_server by specifying its url.
If the hardware is connected to a local machine, -url option and the <url>
are not needed. connect command returns the channel ID of the connection

xsdb% connect -url TCP:xhdbfarmc7:3121 -symbols
tcfchan#0

# List available targets and select a target through its id.
The targets are assigned IDs as they are discovered on the Jtag chain,
so the IDs can change from session to session.
For non-interactive usage, -filter option can be used to select a target,
instead of selecting the target through its ID

xsdb% targets
  1  PS TAP
  2  PMU
      3  MicroBlaze PMU (Sleeping. No clock)
  4  PL
  5  PSU
  6  RPU (Reset)
      7  Cortex-R5 #0 (RPU Reset)
      8  Cortex-R5 #1 (RPU Reset)
  9  APU (L2 Cache Reset)
 10  Cortex-A53 #0 (APU Reset)
 11  Cortex-A53 #1 (APU Reset)
 12  Cortex-A53 #2 (APU Reset)
 13  Cortex-A53 #3 (APU Reset)
xsdb% targets 5

# Configure the FPGA. When the active target is not a FPGA device,
the first FPGA device is configured

xsdb% fpga ZCU102_HwPlatform/design_1_wrapper.bit
100%    36MB    1.8MB/s  00:24
```

```
# Source the psu_init.tcl script and run psu_init command to initialize PS
xsdb% source ZCU102_HwPlatform/psu_init.tcl
xsdb% psu_init

# PS-PL power isolation must be removed and PL reset must be toggled,
before the PL address space can be accessed

# Some delay is needed between these steps

xsdb% after 1000
xsdb% psu_ps_pl_isolation_removal
xsdb% after 1000
xsdb% psu_ps_pl_reset_config

# Select A53 #0 and clear its reset

# To debug 32 bit applications on A53, A53 core must be configured
to boot in 32 bit mode, before the resets are cleared

# 32 bit mode can be enabled through CONFIG_0 register in APU module.
See ZynqMP TRM for details about this register

xsdb% targets 10
xsdb% rst -processor

# Download the application program

xsdb% dow dhystone/Debug/dhystone.elf
Downloading Program -- dhystone/Debug/dhystone.elf
    section, .text: 0xffffc0000 - 0xffffd52c3
    section, .init: 0xffffd5300 - 0xffffd5333
    section, .fini: 0xffffd5340 - 0xffffd5373
    section, .note.gnu.build-id: 0xffffd5374 - 0xffffd5397
    section, .rodata: 0xffffd5398 - 0xffffd6007
    section, .rodata1: 0xffffd6008 - 0xffffd603f
    section, .data: 0xffffd6040 - 0xffffd71ff
    section, .eh_frame: 0xffffd7200 - 0xffffd7203
    section, .mmu_tbl0: 0xffffd8000 - 0xffffd800f
    section, .mmu_tbl1: 0xffffd9000 - 0xffffdafff
    section, .mmu_tbl2: 0xffffdb000 - 0xffffdefff
    section, .init_array: 0xffffdf000 - 0xffffdf007
    section, .fini_array: 0xffffdf008 - 0xffffdf047
    section, .sdata: 0xffffdf048 - 0xffffdf07f
    section, .bss: 0xffffdf080 - 0xffffe197f
    section, .heap: 0xffffe1980 - 0xffffe397f
    section, .stack: 0xffffe3980 - 0xffffe697f
100%    0MB    0.4MB/s  00:00
Setting PC to Program Start Address 0xffffc0000
Successfully downloaded dhystone/Debug/dhystone.elf

# Set a breakpoint at main()
xsdb% bpadd -addr &main
0

# Resume the processor core
xsdb% con

# Info message is displayed when the core hits the breakpoint
Info: Cortex-A53 #0 (target 10) Running
xsdb% Info: Cortex-A53 #0 (target 10) Stopped at 0xffffc0d5c (Breakpoint)

# Registers can be viewed when the core is stopped
```

```
xsdb% rrd
  r0: 0000000000000000      r1: 0000000000000000      r2: 0000000000000000
  r3: 0000000000000004      r4: 000000000000000f      r5: 00000000ffffffff
  r6: 0000000000000001c     r7: 0000000000000002      r8: 00000000ffffffff
  r9: 0000000000000000      r10: 0000000000000000     r11: 0000000000000000
  r12: 0000000000000000     r13: 0000000000000000     r14: 0000000000000000
  r15: 0000000000000000     r16: 0000000000000000     r17: 0000000000000000
  r18: 0000000000000000     r19: 0000000000000000     r20: 0000000000000000
  r21: 0000000000000000     r22: 0000000000000000     r23: 0000000000000000
  r24: 0000000000000000     r25: 0000000000000000     r26: 0000000000000000
  r27: 0000000000000000     r28: 0000000000000000     r29: 0000000000000000
  r30: 0000000fffc1f4c     sp: 0000000ffe5980      pc: 0000000fffc0d5c
cpsr:          600002cd     vfp      sys

# Local variables can be viewed
xsdb% locals
Int_1_Loc      : 1113232
Int_2_Loc      : 30
Int_3_Loc      : 0
Ch_Index       : 0
Enum_Loc       : 0
Str_1_Loc      : char[31]
Str_2_Loc      : char[31]
Run_Index      : 1061232
Number_Of_Runs : 2

# Local variable value can be modified
xsdb% locals Number_Of_Runs 100
xsdb% locals Number_Of_Runs
Number_Of_Runs : 100

# Global variables and be displayed, and its value can be modified
xsdb% print Int_Glob
Int_Glob : 0
xsdb% print -set Int_Glob 23
xsdb% print Int_Glob
Int_Glob : 23

# Expressions can be evaluated and its value can be displayed
xsdb% print Int_Glob + 1 * 2
Int_Glob + 1 * 2 : 25

# Step over a line of source code
xsdb% nxt
Info: Cortex-A53 #0 (target 10) Stopped at 0xffffc0d64 (Step)

# View stack trace
xsdb% bt
  0 0xffffc0d64 main()+8: ../../src/dhry_1.c, line 79
  1 0xffffc1f4c _startup()+84: xil-crt0.S, line 110
```

Note: If the .elf file is not accessible from the remote machine on which the server is running, the xsdb% connect -url TCP:xhdbfarmc7:3121 command should be appended with the -symbols option as shown in the above example.

Selecting Target Based on Target Properties

The following is an example XSCT session that demonstrates selecting a target based on target properties. It shows how to connect to the Cortex®-A9 processor of the second device when multiple devices are connected in a JTAG chain (xc7z020 and xc7z045).

```
# connect to hw_server
xsdb% conn -ho xhdbfarmrkh1
tcfchan#0
# check the jtag targets connected, the IDs listed with jtag targets are
# called node IDs
xsdb% jtag targets
1 Platform Cable USB II 0000153f74cd01
2 arm_dap (idcode 4ba00477 irlen 4)
3 xc7z020 (idcode 03727093 irlen 6 fpga)
4 arm_dap (idcode 4ba00477 irlen 4)
5 xc7z045 (idcode 03731093 irlen 6 fpga)
# check the targets connected, the IDs listed with targets are called
# target IDs
xsdb% targets
1 APU
2 ARM Cortex-A9 MPCore #0 (Suspended)
3 ARM Cortex-A9 MPCore #1 (Suspended)
4 xc7z020
5 APU
6 ARM Cortex-A9 MPCore #0 (Running)
7 ARM Cortex-A9 MPCore #1 (Running)
8 xc7z045
# check jtag target properties of 2nd device (2nd ARM DAP). Note the
# target_ctx here.
xsdb% jtag targets -target-properties -filter {node_id == 4}
{target_ctx jsn-DLC10-0000153f74cd01-4ba00477-1 level 1 node_id 4 is_open 1
is_active 1 is_current 1 name arm_dap jtag_cable_name {Platform Cable USB
II 0000153f74cd01} state {} jtag_cable_manufacturer Xilinx
jtag_cable_product DLC10 jtag_cable_serial 0000153f74cd01 idcode 4ba00477
irlen 4}
# using the target context, select the targets associated with the JTAG
target (2nd ARM DAP - node id = 4)
xsdb% targets -filter {jtag_device_ctx == "jsn-
DLC10-0000153f74cd01-4ba00477-1"}
5 APU
6 ARM Cortex-A9 MPCore #0 (Running)
7 ARM Cortex-A9 MPCore #1 (Running)
```

Memory and Register accesses from XSCT

Memory accesses

Memory accesses in XSCT take different physical paths in the SoC devices based on the active target.

Processor targets

When a processor is selected as an active target, any memory read/write commands (mrd/mwr) run by the users are executed by the processor. The debugger injects load/store instructions into the processor to access the memory. Because the processor executes these instructions, MMU and caches come into the picture. The address of the read/write commands is treated as virtual address by the processor. Data is read from or written to caches. If MMU and caches are disabled, physical memory is accessed during load/store.

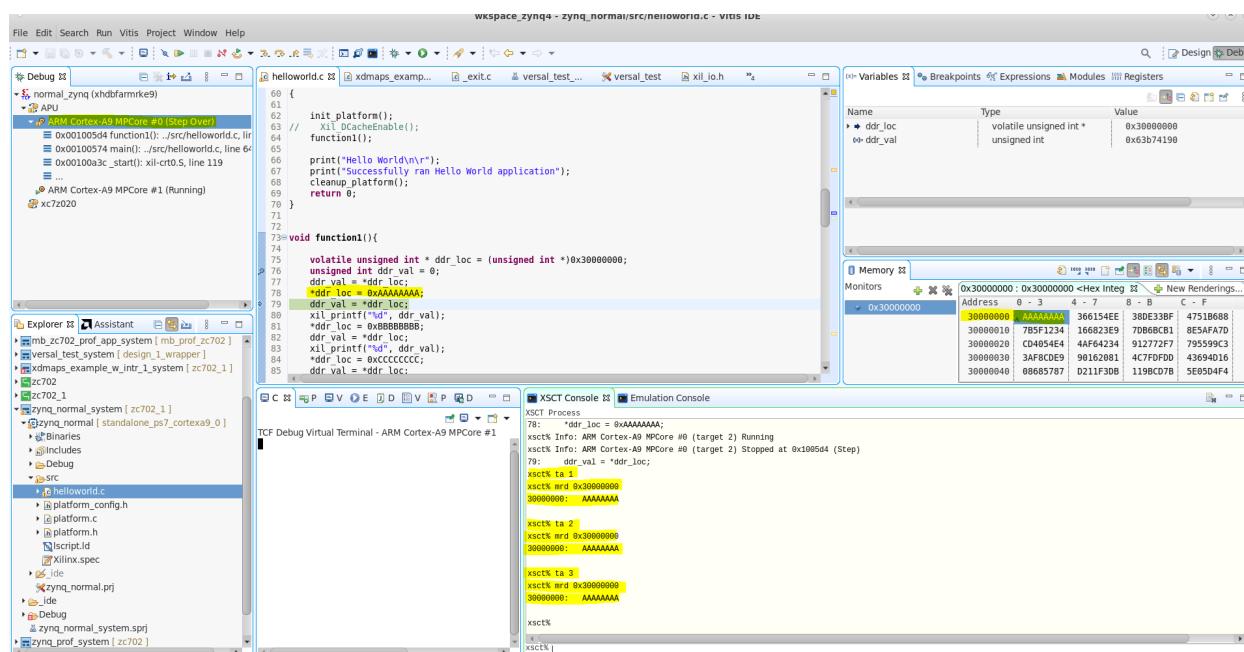
Example, processor targets are Cortex-A9, Cortex®-A53, Cortex-A72, and so on.

Non-processor targets

When targets like APU/RPU/PSU/Versal are selected as active targets, physical memory is accessed during memory read/write commands. These targets use AXI-AP in Arm® DAP to access memory. The AXI-AP does not have access to the MMU or caches inside processor targets. However, the debugger flushes/invalidates the caches for every memory access command. Therefore, it is the same data on any target, APU/processor, even though the cache is enabled. Following are the examples.

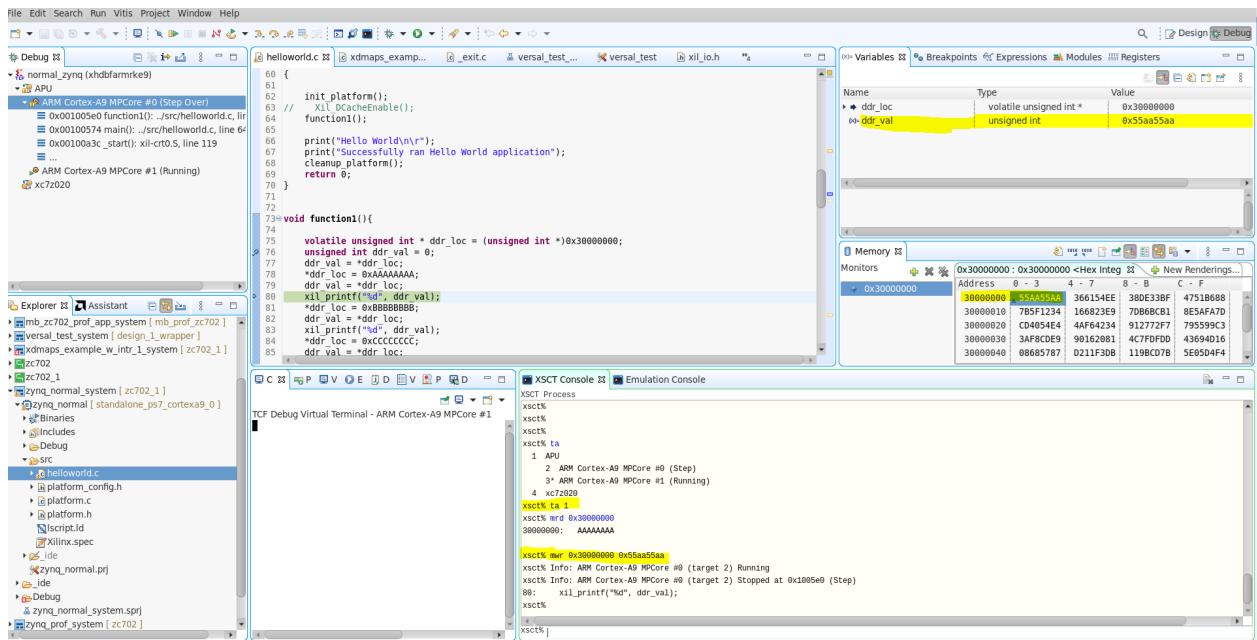
Case 1: Write 0xAAAAAAA to the location 0x30000000 from the processor target and read it from the APU target or processor target. The data is the same.

Figure 68: Behavior of Memory-writes from Processor Target



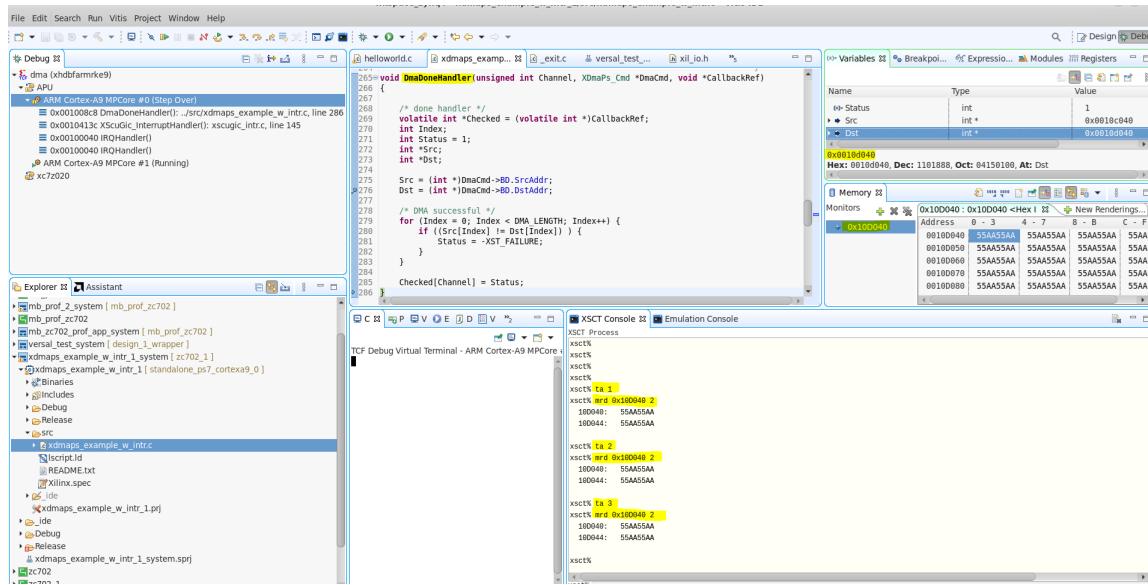
Case 2: Write 0x55AA55AA to the location 0x30000000 from the APU target and read it from the APU target or processor target. The data is the same in both targets. In the below screenshot, select target 1 on Zynq, which is APU, and write 0x55AA55AA. From Vitis, check the ddr_val variable from the processor target, which is updated as soon as we write from target1.

Figure 69: Behavior of Memory-writes from APU Target



Case 3: Perform DMA transfers from the application (processor target) by filling source location (0x10C040) with 0x55AA55AA, of size 1024 words, and destination location (0x10D040) with 0xDEADBEEF, of size 1024 words. Initiate the DMA transfer. After the transfer is done, verify the destination location from the processor and APU Targets. The data will be the same.

Figure 70: Behavior of DMA-transfers from Processor Target



The Effect of XPPU/XMPU on Memory Accesses

If XMPU/XPPU are configured to block accesses to a peripheral/memory through a processor or DAP, accessing such addresses thru these targets in XSCT will lead to memory access errors. Memory accesses succeed only when the hardware is configured to allow access from the active target to that address. Users must use 'loadhw' or 'memmap' commands that will provide access to the protected memory regions. Alternatively, '-force' option in 'mrd'/'mwr' commands can be used or set 'force-mem-access' in 'configparams' to 1.

Register Accesses

The registers list varies based on the active target. For processor targets, the registers list includes processor registers like general purpose register, PC, SP, LR, special registers, banked registers and so on.

For non-processor targets like APU/RPU/PSU/Versal, the registers list includes registers from all the peripherals in the SoC. For example, peripherals include DDRC, UART, SPI and so on.

Registers are grouped wherever applicable. Such registers can be accessed by running mrd/mwr
<register-group> <register-name> ?value?

Modifying BSP Settings

Below is an example XSCT session that demonstrates building a HelloWorld application to target the MicroBlaze™ processor. The STDIN and STDOUT OS parameters are changed to use the MDM_0.

Note: When the BSP settings are changed, it is necessary to update the mss and regenerate the BSP sources to reflect the changes in the source file before compiling.

```
setws /tmp/wrk/workspace
app create -name mb_app -hw /tmp/wrk/kc705_system.xsa -proc microblaze_0 -
os standalone -template {Hello World}
bsp config stdin mdm_0
bsp config stdout mdm_0
bsp regenerate
platform generate
app build -name mb_app

setws /tmp/wrk/workspace
app create -name mb_app -hw /tmp/wrk/kc705_system.xsa -proc microblaze_0 -
os standalone -template

{Hello World}
bsp config stdin mdm_0
bsp config stdout mdm_0
bsp regenerate
platform generate
app build -name mb_app
```

Performing Standalone Application Debug

System Command-line Tool (XSCT) can be used to debug standalone applications on one or more processor cores simultaneously. The first step involved in debugging is to connect to hw_server and select a debug target. You can now reset the system/processor core, initialize the PS if needed, program the FPGA, download an elf, set breakpoints, run the program, examine the stack trace, view local/global variables.

An example XSCT session that demonstrates standalone application debug on AMD Zynq™ 7000 SoC is as follows. Comments begin with #.

```
#connect to remote hw_server by specifying its url.
#If the hardware is connected to a local machine,-url option and the <url>
#are not needed. connect command returns the channel ID of the connection

xsct% connect -url TCP:xhdbfarmc7:3121 tcfchan#0

# List available targets and select a target through its id.
#The targets are assigned IDs as they are discovered on the Jtag chain,
#so the IDs can change from session to session.
#For non-interactive usage, -filter option can be used to select a target,
```

```
#instead of selecting the target through its ID

xsct% targets
 1 APU
 2 Arm Cortex-A9 MPCore #0 (Running)
 3 Arm Cortex-A9 MPCore #1 (Running)
 4 xc7z020
xsct% targets 2
# Reset the system before initializing the PS and configuring the FPGA

xsct% rst
# Info messages are displayed when the status of a core changes
Info: Arm Cortex-A9 MPCore #0 (target 2) Stopped at 0xffffffffe1c (Suspended)
Info: Arm Cortex-A9 MPCore #1 (target 3) Stopped at 0xffffffffe18 (Suspended)

# Configure the FPGA. When the active target is not a FPGA device,
#the first FPGA device is configured

xsct% fpga ZC702_HwPlatform/design_1_wrapper.bit
100% 3MB 1.8MB/s 00:02

# Run loadhw command to make the debugger aware of the processor cores'
memory map
xsct% loadhw ZC702_HwPlatform/system.xsa
design_1_wrapper

# Source the ps7_init.tcl script and run ps7_init and ps7_post_config
commands
xsct% source ZC702_HwPlatform/ps7_init.tcl
xsct% ps7_init
xsct% ps7_post_config

# Download the application program
xsct% dow dhystone/Debug/dhystone.elf
Downloading Program -- dhystone/Debug/dhystone.elf
  section, .text: 0x00100000 - 0x001037f3
  section, .init: 0x001037f4 - 0x0010380b
  section, .fini: 0x0010380c - 0x00103823
  section, .rodata: 0x00103824 - 0x00103e67
  section, .data: 0x00103e68 - 0x001042db
  section, .eh_frame: 0x001042dc - 0x0010434f
  section, .mmu_tbl: 0x00108000 - 0x0010bfff
  section, .init_array: 0x0010c000 - 0x0010c007
  section, .fini_array: 0x0010c008 - 0x0010c00b
  section, .bss: 0x0010c00c - 0x0010e897
  section, .heap: 0x0010e898 - 0x0010ec9f
  section, .stack: 0x0010eca0 - 0x0011149f
100% 0MB 0.3MB/s 00:00

Setting PC to Program Start Address 0x00100000

Successfully downloaded dhystone/Debug/dhystone.elf

# Set a breakpoint at main()
xsct% bpadd -addr &main
0

# Resume the processor core
xsct% con

# Info message is displayed when the core hits the breakpoint
xsct% Info: Arm Cortex-A9 MPCore #0 (target 2) Stopped at 0x1005a4
(Breakpoint)
```

```
# Registers can be viewed when the core is stopped
xsct% rrd
    r0: 00000000      r1: 00000000      r2: 0010e898      r3: 001042dc
    r4: 00000003      r5: 0000001e      r6: 0000ffff      r7: f8f00000
    r8: 00000000      r9: ffffffff      r10: 00000000     r11: 00000000
    r12: 0010fc90     sp: 0010fcfa0     lr: 001022d8      pc: 001005a4
    cpsr: 600000df     usr             fiq             irq
    abt              und             svc             mon
    vfp              cp15            Jazelle

# Memory contents can be displayed
xsct% mrd 0xe000d000
E000D000: 800A0000

# Local variables can be viewed
xsct% locals
Int_1_Loc      : 1113232
Int_2_Loc      : 30
Int_3_Loc      : 0
Ch_Index       : 0
Enum_Loc       : 0
Str_1_Loc      : char[31]
Str_2_Loc      : char[31]
Run_Index      : 1061232
Number_Of_Runs : 2

# Local variable value can be modified
xsct% locals Number_Of_Runs 100
xsct% locals Number_Of_Runs
Number_Of_Runs : 100

# Global variables can be displayed, and its value can be modified
xsct% print Int_Glob
Int_Glob : 0
xsct% print -set Int_Glob 23
xsct% print Int_Glob
Int_Glob : 23

# Expressions can be evaluated and its value can be displayed
xsct% print Int_Glob + 1 * 2
Int_Glob + 1 * 2 : 25

# Step over a line of source code
xsct% nxt
Info: Arm Cortex-A9 MPCore #0 (target 2) Stopped at 0x1005b0 (Step)

# View stack trace
xsct% bt
  0 0x1005b0 main()+12: ../src/dhry_1.c, line 91
  1 0x1022d8 _start()+88
  2 unknown-pc

# Set a breakpoint at exit and resume execution
xsct% bpadd -addr &exit
1
xsct% con
Info: Arm Cortex-A9 MPCore #0 (target 2) Running
xsct% Info: Arm Cortex-A9 MPCore #0 (target 2) Stopped at 0x103094
```

```
(Breakpoint)
xsct% bt
 0 0x103094 exit()
 1 0x1022e0 __start() +96
 2 unknown -pc
```

While a program is running on A9 #0, you can download another elf onto A9 #1 and debug it, using similar steps. It is not necessary to re-connect to the hw_server, initialize the PS or configure the FPGA in such cases. You can select A9 #1 target and download the elf and continue with further debug.

Generating SVF Files

SVF (Serial Vector Format) is an industry standard file format that is used to describe JTAG chain operations in a compact, portable fashion. An example XSCT script to generate an SVF file is as follows:

```
# Reset values of respective cores
set core 0
set apu_reset_a53 {0x380e 0x340d 0x2c0b 0x1c07}
# Generate SVF file for linking DAP to the JTAG chain
# Next 2 steps are required only for Rev2.0 silicon and above.
svf config -scan-chain {0x14738093 12 0x5ba00477 4
} -device-index 1 -linkdap -out "dapcon.svf"
svf generate
# Configure the SVF generation
svf config -scan-chain {0x14738093 12 0x5ba00477 4
} -device-index 1 -cpu-index $core -delay 10 -out "fsbl_hello.svf"
# Record writing of bootloop and release of A53 core from reset
svf mwr 0xfffff0000 0x14000000
svf mwr 0xfd1a0104 [lindex $apu_reset_a53 $core]
# Record stopping the core
svf stop
# Record downloading FSBL
svf dow "fsbl.elf"
# Record executing FSBL
svf con
svf delay 100000
# Record some delay and then stopping the core
svf stop
# Record downloading the application
svf dow "hello.elf"
# Record executing application
svf con
# Generate SVF
svf generate
```

Note: SVF files can only be recorded using XSCT. You can use any standard SVF player to play the SVF file.

To play a SVF file in the Vivado hardware manager, connect to a target and use the following Tcl command to play the file on the selected target.

```
execute_hw_svf <*.svf file>
```

Program U-BOOT over JTAG

```
#connecting to the target
connect -host <hostname>

#Disable security gates to view PMU MB target
targets -set -filter {name =~ "PSU"}
;
#By default, JTAG security gates are enabled
#This disables security gates for DAP, PLTAP, and PMU.
mwr 0xffca0038 0x1ff
after 500
;
#Load and run PMU FW
targets -set -filter {name =~ "MicroBlaze PMU"}5)
dow pmufw.elf
con
after 500
;
#Reset A53, load and run FSBL
targets -set -filter {name =~ "Cortex-A53 #0"}
rst -processor
dow zynqmp_fsbl.elf
con
#Give FSBL time to run
after 5000
stop
;
#Downloading other Softwares like U-boot ans so on, using below command
dow -data system.dtb 0x100000
dow u-boot.elf
dow b131.elf
con
after 5000
stop
```

Running an Application in Non-Interactive Mode

Xilinx System Debugger Command-line Interface (XSDB) provides a scriptable interface to run applications in non-interactive mode. To run the program in previous example using a script, create a Tcl script (and name it as, for example, `test.tcl`) with the following commands. The script can be run by passing it as a launch argument to XSDB.

```
connect -url TCP:xhdbfarmc7:3121

# Select the target whose name starts with Arm and ends with #0.
# On Zynq, this selects "Arm Cortex-A9 MPCore #0"

targets -set -filter {name =~ "Arm* #0"}
rst
fpga ZC702_HwPlatform/design_1_wrapper.bit
```

```
loadhw ZC702_HwPlatform/system.xsa
source ZC702_HwPlatform/ps7_init.tcl
ps7_init
ps7_post_config
dow dhystone/Debug/dhystone.elf

# Set a breakpoint at exit

bpadd -addr &exit

# Resume execution and block until the core stops (due to breakpoint)
# or a timeout of 5 sec is reached

con -block -timeout 5
```

Running Tcl Scripts

You can create Tcl scripts with XSCT commands and run them in an interactive or non-interactive mode. In the interactive mode, you can source the script at XSCT prompt. For example:

```
xsct% source xsct_script.tcl
```

In the non-interactive mode, you can run the script by specifying the script as a launch argument. Arguments to the script can follow the script name. For example:

```
$ xsct xsct_script.tcl [args]
```

The script below provides a usage example of XSCT. This script creates and builds an application, connects to a remote hw_server, initializes the Zynq PS connected to remote host, downloads and executes the application on the target. These commands can be either scripted or run interactively.

```
# Set Vitis workspace
setws /tmp/workspace
# Create application project
app create -name hello -hw /tmp/wrk/system.xsa -proc ps7_cortexa9_0 -os
standalone -lang C -template {Hello World}
app build -name hello hw_server
connect -host raptor-host
# Select a target
targets -set -nocase -filter {name =~ "Arm* #0"}
# System Reset
rst -system
# PS7 initialization
namespace eval xsdb {source /tmp/workspace/hw1/ps7_init.tcl; ps7_init}
# Download the elf
dow /tmp/workspace/hello/Debug/hello.elf
# Insert a breakpoint @ main
bpadd -addr &main
# Continue execution until the target is suspended
```

```
con -block -timeout 500
# Print the target registers
puts [rrd]
# Resume the target
con
```

Switching Between XSCT and Vitis Integrated Design Environment

Below is an example XSCT session that demonstrates creating and building an application using XSCT. After execution, launch the Vitis development environment and select the workspace created using XSCT to view the updates.

Note: The workspace created in XSCT can be used from Vitis IDE. However, at a time, only one instance of the tool can use the workspace.

```
# Set Vitis workspace
setws /tmp/workspace
# Create application project
app create -name hello -hw /tmp/wrk/system.xsa -proc ps7_cortexa9_0 -os
standalone -lang C -template {Hello World}
app build -name hello
```

Using JTAG UART

XSDB supports virtual UART through JTAG, which is useful when the physical UART does not exist or is non-functional. To use JTAG UART, the software application should be modified to redirect STDIO to the JTAG UART. Vitis IDE provides a CoreSight™ driver to support redirecting of STDIO to virtual UART on Arm based designs. For MB designs, the uartlite driver can be used. To use the virtual UART driver, open board support settings in Vitis IDE and can change STDIN / STDOUT to coresight/mdm.

XSDB supports virtual UART through two commands.

- **jtagterminal** - Start/Stop JTAG based hyper-terminal. This command opens a new terminal window for STDIO. The text input from this terminal will be sent to STDIN and any output from STDOUT will be displayed on this terminal.
- **readjtaguart** - Start/Stop reading from JTAG UART. This command starts polling STDOUT for output and displays in on XSDB terminal or redirects it to a file.

An example XSCT session that demonstrates how to use a JTAG terminal for STDIO is as follows:

```
connect
source ps7_init.tcl
targets -set -filter {name =~ "APU"}
loadhw system.xsa
stop
ps7_init
targets -set -nocase -filter {name =~ "Arm*#0"}
rst -processor
dow <app>.elf
jtagterminal
con
jtagterminal -stop #after you are done
```

An example XSCT session that demonstrates how to use the XSCT console as STDOUT for JTAG UART is as follows:

```
connect
source ps7_init.tcl
targets -set -filter {name =~ "APU"}
loadhw system.xsa
stop
ps7_init
targets -set -nocase -filter {name =~ "Arm*#0"}
rst -processor
dow <app>.elf
readjtaguart
con
readjtaguart -stop #after you are done
```

An example XSCT session that demonstrates how to redirect the STDOUT from JTAG UART to a file is as follows:

```
connect
source ps7_init.tcl
targets -set -filter {name =~ "APU"}
loadhw system.xsa
stop
ps7_init
targets -set -nocase -filter {name =~ "Arm*#0"}
rst -processor
dow <app>.elf
set fp [open uart.log w]
readjtaguart -handle $fp
con
readjtaguart -stop #after you are done
```

Working with Libraries

An example XSCT session that demonstrates creating a default domain and adding XILFFS and XILRSA libraries to the BSP is as follows. Create a FSBL application thereafter.

Note: A normal domain/BSP does not contain any libraries.

```
setws /tmp/wrk/workspace
app create -name hello -hw /tmp/wrk/system.xsa -proc ps7_cortexa9_0 -os
standalone -lang C -template {Hello World}
bsp setlib -name xilffs
bsp setlib -name xilrsa
platform generate
app build -name hello
```

Changing the OS version.

```
bsp setosversion -ver 6.6
```

Assigning a driver to an IP.

```
bsp setdriver -ip ps7_uart_1 -driver generic -ver 2.0
```

Removing a library (removes xilrsa library from the domain/BSP).

```
bsp removelib -name xilrsa
```

Editing FSBL/PMUFW Source File

The following example shows you how to edit FSBL/PMUFW source files.

```
setws workspace
app create -name a53_app -hw zcu102 -os standalone -proc psu_cortexa53_0
#Go to "workspace/zcu102/zynqmp_fsbl" or "workspace/zcu102/zynqmp_pmufw"
and modify the source files using any editor like gedit or gvim for boot
domains zynqmp_fsbl and zynqmp_pmufw.
platform generate
```

Editing FSBL/PMUFW Settings

The following example shows you how to edit FSBL/PMUFW settings.

```
setws workspace
app create -name a53_app -hw zcu102 -os standalone -proc psu_cortexa53_0
#If you want to modify anything in zynqmp_fsbl domain use below command to
active that domain
domain active zynqmp_fsbl
#If you want to modify anything in zynqmp_pmufw domain use below command to
active that domain
domain active zynqmp_pmufw
#configure the BSP settings for boot domain like FSBL or PMUFW
bsp config -append compiler_flags -DFSBL_DEBUG_INFO
platform generate
```

Exchanging Files between Host Machine and Linux Running on QEMU

XSCT `tfile` can be used to communicate with the tcf-agent running in Linux to transfer files. To exchange file between host machine and Linux in QEMU, follow these steps:

1. Launch QEMU from the Vitis IDE by selecting **Vitis → Start/Stop Emulator**. QEMU is launched to boot Linux. The tcf-agent runs in the backend when Linux finishes booting. It is required to include the tcf-agent in the Linux root file system configuration in PetaLinux.
2. Launch XSCT and use the following commands to exchange file:
 - a. Connect to the tcf-agent using XSCT:

```
connect -host 127.0.0.1 -port 1440
```

Note: 1440 is port forwarded by QEMU.

- b. Copy file from host to target:

```
tfile copy -from-host <host_path> <target_path>
```

- c. Copy file from target to host:

```
tfile copy -to-host <target_path> <host_path>
```

Loading U-Boot over JTAG

Users can load and run U-Boot over JTAG and then flash drivers in the U-Boot to program the flash.

Script to run U-Boot and download the binary file for programming to flash.

- Zynq:

```
# Connect to target
connect

# Load and run FSBL
targets -set -nocase -filter {name =~ "ARM*#0"}
dow zynq_fsbl.elf
con
after 3000
stop

# Load DTB at 0x100000
dow -data system.dtb 0x100000

# Download and run u-boot
dow uboot.elf
```

```
con
after 1000;
stop

# Download the BOOT.bin (file to program to flash) in some DDR location
# which is not used for other apps.
dow -data BOOT.BIN <ddr_addr>
```

- AMD Zynq™MP:

```
# Connect to target
connect

# Disable Security gates to view PMU MB target
targets -set -nocase -filter {name =~ "*PSU*"}
mask_write 0xFFCA0038 0x1C0 0x1C0

# Load and run PMU FW
targets -set -nocase -filter {name =~ "*MicroBlaze PMU*"}
dow pmufw.elf
con
after 500

# Load and run FSBL
targets -set -nocase -filter {name =~ "*A53*#0"}
rst -proc
dow zynqmp_fsbl.elf
con
after 5000
stop

# Load DTB at 0x100000
dow -data system.dtb 0x100000

# Load and run u-boot
dow u-boot.elf
dow bl31.elf
con
after 5000
stop

# Download the BOOT.bin (file to program to flash) in some DDR location
# which is not used for other apps.
dow -data BOOT.BIN <ddr_address>
```

- Versal:

```
# Connect to target
connect

# Configure the device with PDI containing PLM, necessary CDOs, u-boot,
BL31 and system.dtb
# Steps for creating this PDI (BOOT.BIN) are given in the next section
targets -set -nocase -filter {name =~ "*PMC*"}
device program BOOT.BIN

# Download the BOOT.bin (file to program to flash) in some DDR location
# which is not used for other apps.
targets -set -nocase -filter {name =~ "*A72*#0"}
stop
dow -data BOOT.BIN <ddr_address>
```

Following are the steps to create BOOT.BIN used in the script:

1. Extract the PDI from the XSA - lets call it system.pdi
2. Create a BIF as shown below: bootimage.bif

```
all:
{
    image
    {
        { type = bootimage, file = system.pdi }
    }
    image
    {
        name=default_subsys, id=0x1c000000
        { load = 0x1000, file = system.dtb }
        { core = a72-0, exception_level = el-3, trustzone, file =
b131.elf }
        { core = a72-0, exception_level = el-2, load=0x8000000, file=u-
boot.elf }
    }
}
```

3. Use Bootgen to create a new extended PDI by appending system.dtb, U-Boot and b131 to the PDI extracted from XSA Bootgen -arch versal -image bootimage.bif -w -o BOOT.BIN.

Running U-Boot commands for flash programming

After loading and running U-Boot, at the U-Boot console, input the following commands:

```
sf probe 0 0 0
sf erase 0 <size of BOOT.bin>
sf write <ddr_address> 0 <size of BOOT.bin>
```

Hardware Software Interface (HSI) Commands

XSCT Interface Examples

HSI Tcl Examples

This chapter demonstrates how to load an .xsa file, access the hardware information, and generate BSPs, applications, and the Device Tree.

Accessing Hardware Design Data

Opening the hardware design

```
hsi::open_hw_design base_zynq_design_wrapper.xsa  
base_zynq_design_imp
```

List loaded hardware designs

```
hsi::get_hw_designs  
base_zynq_design_imp
```

Switch to current hardware design

```
hsi::current_hw_design  
base_zynq_design_imp
```

Report properties of the current hardware design

```
common::report_property [hsi::current_hw_design]
```

Table 53: Example Table

Property	Type	Read Only	Visible	Value
ADDRESS_TAG	string*	true	true	base_zynq_design_i/ ps7_cortexa9_0:base_zynq_design_i base_zynq_design_i/ ps7_cortexa9_1:base_zynq_design_i
BOARD	string	true	true	xilinx.com:zc702:part0:1.1
CLASS	string	true	true	hw_design
DEVICE	string	true	true	7x020
FAMILY	string	true	true	zynq
NAME	string	true	true	base_zynq_design_imp
PACKAGE	string	true	true	clg484
PATH	string	true	true	/scratch/demo//base_zynq_design.hwh
SPEEDGRADE	string	true	true	1
SW_REPOSITORIES	string*	true	true	
TIMESTAMP	string	true	true	<current date and time>
VIVADO_VERSION	string	true	true	2014.3

List the .xsa files in the container

```
hsi::get_hw_files
base_zynq_design.hwh ps7_init.c ps7_init.h ps7_init_gpl.c
ps7_init_gpl.h ps7_init.tcl ps7_init.html
base_zynq_design_wrapper.mmi base_zynq_design_bd.tcl
```

Filter the .bit files

```
hsi::get_hw_files -filter {TYPE==bit}
base_zynq_design_wrapper.bit
```

List of external ports in the design

```
hsi::get_ports
DDR_cas_n DDR_cke DDR_ck_n DDR_ck_p DDR_cs_n DDR_reset_n
DDR_odt DDR_ras_n
DDR_we_n DDR_ba DDR_addr DDR_dm DDR_dq DDR_dqs_n DDR_dqs_p
FIXED_IO_mio
FIXED_IO_ddr_vrn FIXED_IO_ddr_vrp FIXED_IO_ps_srstb
FIXED_IO_ps_clk
FIXED_IO_ps_porb leds_4bits_tri_o
```

Reports properties of an external port

```
common::report_property [hsi::get_ports leds_4bits_tri_o]
```

Table 54: Example Table

Property	Type	Readonly	Visible	Value
CLASS	string	true	true	port
CLK_FREQ	string	true	true	
DIRECTION	string	true	true	0
INTERFACE	bool	true	true	0
IS_CONNECTED	bool	true	true	0
LEFT	string	true	true	3
NAME	string	true	true	leds_4bits_tri_o
RIGHT	string	true	true	0
SENSITIVITY	enum	true	true	
TYPE	enum	true	true	undef

List of IP instances in the design

```
hsi::get_cells
axi_bram_ctrl_0 axi_gpio_0 blk_mem_gen_0
processing_system7_0_axi_periph_m00_couplers_auto_pc
processing_system7_0_axi_periph_s00_couplers_auto_pc
processing_system7_0_axi_periph_xbar
rst_processing_system7_0_50M ps7_clockc_0 ps7_uart_1
ps7_p1310_0 ps7_pmu_0 ps7_qspi_0
ps7_qspi_linear_0 ps7_axi_interconnect_0 ps7_cortexa9_0
ps7_cortexa9_1 ps7_ddr_0
ps7_ethernet_0 ps7_usb_0 ps7_sd_0 ps7_i2c_0 ps7_can_0
ps7_ttc_0 ps7_gpio_0
ps7_ddrc_0 ps7_dev_cfg_0 ps7_xadc_0 ps7_ocmc_0
ps7_coresight_comp_0 ps7_gpv_0 ps7_scuc_0
ps7_globaltimer_0 ps7_intc_dist_0 ps7_l2cachec_0 ps7_dma_s
ps7_iop_bus_config_0 ps7_ram_0
ps7_ram_1 ps7_scugic_0 ps7_scutimer_0 ps7_scuwdt_0
ps7_s1lcr_0 ps7_dma_ns ps7_afi_0 ps7_afi_1
ps7_afi_2 ps7_afi_3 ps7_m_axi_gp0
```

#List of processors in the design

```
hsi::get_cells -filter {IP_TYPE==PROCESSOR}
ps7_cortexa9_0 ps7_cortexa9_1
```

Properties of IP instance

```
common::report_property [hsi::get_cells axi_gpio_0]
```

Table 55: Example Table

Property	Type	Readonly	Visible	Value
CLASS	string	true	true	cell
CONFIG.C_ALL_INPUTS	string	true	true	0
CONFIG.C_ALL_INPUTS_2	string	true	true	0

Table 55: Example Table (cont'd)

Property	Type	Readonly	Visible	Value
CONFIG.C_ALL_OUTPUTS	string	true	true	1
CONFIG.C_ALL_OUTPUTS_2	string	true	true	0
CONFIG.C_BASEADDR	string	true	true	0x41200000
CONFIG.C_DOUT_DEFAULT	string	true	true	0x00000000
CONFIG.C_DOUT_DEFAULT_2	string	true	true	0x00000000
CONFIG.C_FAMILY	string	true	true	zynq
CONFIG.C_GPIO2_WIDTH	string	true	true	32
CONFIG.C_GPIO_WIDTH	string	true	true	4
CONFIG.C_HIGHADDR	string	true	true	0x4120FFFF
CONFIG.C_INTERRUPT_PRESENT	string	true	true	0
CONFIG.C_IS_DUAL	string	true	true	0
CONFIG.C_S_AXI_ADDR_WIDTH	string	true	true	9
CONFIG.C_S_AXI_DATA_WIDTH	string	true	true	32
CONFIG.C_TRI_DEFAULT	string	true	true	0xFFFFFFFF
CONFIG.C_TRI_DEFAULT_2	string	true	true	0xFFFFFFFF
CONFIG.Component_Name	string	true	true	base_zynq_design_axi_gpio_0_0
CONFIG.EDK_IPTYPE	string	true	true	PERIPHERAL
CONFIG.GPIO2_BOARD_INTERFACE	string	true	true	Custom
CONFIG.GPIO_BOARD_INTERFACE	string	true	true	leds_4bits
CONFIG.USE_BOARD_FLOW	string	true	true	true
CONFIGURABLE	bool	true	true	0
IP_NAME	string	true	true	axi_gpio
IP_TYPE	enum	true	true	PERIPHERAL
NAME	string*	true	true	axi_gpio_0
PRODUCT_GUIDE	string	true	true	AXI GPIO LogiCORE IP Product Guide (PG144)
SLAVES	string	true	true	
VNV	string	true	true	xilinx.com:ip:axi_gpio:2.0

Memory range of the Slave IPs

```
common::report_property [lindex [hsi::get_mem_ranges -of_objects
[hsi::get_cells -filter {IP_TYPE==PROCESSOR}]] 39]
```

Table 56: Example Table

Property	Type	Read-only	Visible	Value
BASE_NAME	string	true	true	C_BASEADDR

Table 56: Example Table (cont'd)

Property	Type	Read-only	Visible	Value
BASE_VALUE	string	true	true	0x41200000
CLASS	string	true	true	mem_range
HIGH_NAME	string	true	true	C_HIGHADDR
HIGH_VALUE	string	true	true	0x4120FFFF
INSTANCE	cell	true	true	axi_gpio_0
IS_DATA	bool	true	true	1
IS_INSTRUCTION	bool	true	true	0
MEM_TYPE	enum	true	true	REGISTER
NAME	string	true	true	axi_gpio_0

Creating Standalone Software Design and Accessing Software Information

List of the drivers in the software repository

```
hsim::get_sw_cores *uart*
uartlite_v2_01_a uartlite_v3_0 uartns550_v2_01_a
uartns550_v2_02_a uartns550_v3_0
uartns550_v3_1 uartps_v1_04_a uartps_v1_05_a uartps_v2_0
uartps_v2_1 uartps_v2_2
```

Creates software design

```
hsim::create_sw_design swdesign -proc ps7_cortexa9_0 -os standalone
swdesign
```

To switch to active software design

```
hsim::current_sw_design
swdesign
```

Properties of the current software design

```
common::report_property [hsim::current_sw_design ]
```

Table 57: Example Table

Property	Type	Read-only	Visible	Value
APP_COMPILER	string	FALSE	TRUE	arm-xilinx-eabi-gcc
APP_COMPILER_FLAGS	string	FALSE	TRUE	
APP_LINKER_FLAGS	string	FALSE	TRUE	
BSS_MEMORY	string	FALSE	TRUE	
CLASS	string	TRUE	TRUE	sw_design
CODE_MEMORY	string	FALSE	TRUE	
DATA_MEMORY	string	FALSE	TRUE	

Table 57: Example Table (cont'd)

Property	Type	Read-only	Visible	Value
NAME	string	TRUE	TRUE	swdesign

```
# The drivers associated to current hardware design
```

```
hsi::get_drivers
axi_bram_ctrl_0 axi_gpio_0 ps7_afi_0 ps7_afi_1 ps7_afi_2
ps7_afi_3 ps7_can_0
ps7_coresight_comp_0 ps7_ddr_0 ps7_ddrc_0 ps7_dev_cfg_0
ps7_dma_ns ps7_dma_s
ps7_ethernet_0 ps7_globaltimer_0 ps7_gpio_0 ps7_gpv_0
ps7_i2c_0 ps7_intc_dist_0
ps7_iop_bus_config_0 ps7_l2cachec_0 ps7_ocmc_0 ps7_pl1310_0
ps7_pmu_0 ps7_qspi_0
ps7_qspi_linear_0 ps7_ram_0 ps7_ram_1 ps7_scuc_0
ps7_scugic_0 ps7_scutimer_0
ps7_scuwdt_0 ps7_sd_0 ps7_slcr_0 ps7_ttc_0 ps7_uart_1
ps7_usb_0 ps7_xadc_0
hsি% get-osstandalone
```

```
# Properties of the OS object
```

```
common::report_property[hsi::get_os]
```

Table 58: Example Table

Property	Type	Read-only	Visible	Value
CLASS	string	TRUE	TRUE	sw_proc
CONFIG.archiver	string	FALSE	TRUE	arm-xilinx-eabi-ar
CONFIG.compiler	string	FALSE	TRUE	arm-xilinx-eabi-gcc
CONFIG.compiler_flags	string	FALSE	TRUE	-O2 -c
CONFIG.extra_compile_r_flags	string	FALSE	TRUE	-g
HW_INSTANCE	string	TRUE	TRUE	ps7_cortexa9_0
NAME	string	FALSE	TRUE	cpu_cortexa9
VERSION	string	FALSE	TRUE	2.1

```
# Generate BSP. BSP source code will be dumped to the output directory.
```

```
hsi::generate_bsp -dir bsp_out
```

```
# List of available apps in the repository
```

```
hsi::generate_app -lapp
peripheral_tests dhystone empty_application hello_world
lwip_echo_server
memory_tests rsa_auth_app srec_bootloader
xilkernel_thread_demo zynq_dram_test
zynq_fsbl linux_empty_app linux_hello_world
opencv_hello_world
```

```
# Generate template application
```

```
hsi::generate_app -app hello_world -proc ps7_cortexa9_0 -
dir app_out
```

```
# Generate Device Tree. Clone device tree repo from GIT to /device_tree_repository/
device-treegenerator-master directory.
```

```
# Load the hardware design
```

```
hsi::open_hw_design zynq_1_wrapper.xsa
```

```
# Cloned GIT repo path
```

```
hsi::set_repo_path ./device_tree_repository/device-tree-generator-master
```

```
# Create sw design
```

```
hsi::create_sw_design sw1 -proc ps7_cortexa9_0 -os device_tree
```

```
# Generate device tree
```

```
hsi::generate_target {dts} -dir dtg_out
```

Generating and Compiling Applications with Customized Compiler Settings and Memory Sections

```
#Create a software design for the template application with default compiler flags and memory
section settings
```

```
set sw_system_1 [hsi::create_sw_design system_1 -proc microblaze_1 -os
xilkernel -app hello_world]
```

```
#Change compiler and its flags of the software design
```

```
common::set_property APP_COMPILER "mb-gcc" $sw_system_1
common::set_property -name APP_COMPILER_FLAGS -value "-DRSA_SUPPORT -
DFSBL_DEBUG_INFO"
-objects $sw_system_1
common::set_property -name APP_LINKER_FLAGS -value "-Wl,--start-group,-
lxil,-lgcc,-lc,--end-group"
-objects $sw_system_1
```

#Change memory sections

```
common::set_property CODE_MEMORY axi_bram_ctrl_1 $sw_system_1
common::set_property BSS_MEMORY axi_bram_ctrl_1 $sw_system_1
common::set_property DATA_MEMORY axi_bram_ctrl_2 $sw_system_1
```

#Generate application for the above customized software design to Zynq_Fsbl directory

```
hsi::generate_app -dir hw_output -compile
```

Generating and Compiling BSP with Advanced Driver/Library/OS/Processor Configuration

#Create a software design for the template application with default compiler flags and memory section settings

```
set sw_system_1 [hsi::create_sw_design system_1 -proc microblaze_1 -os xilkernel ]
```

#Get the old driver object

```
set old_driver [hsi::get_drivers myip1]
```

#Set repository path to find the custom drivers and libraries

```
hsi::set_repo_path ./my_local_sw_repository
```

#Set the new driver name and version to old driver object

```
common::set_property NAME myip1_custom_driver $old_driver
common::set_property VERSION 1.0 $old_driver
```

#Change default OS configuration to desired one

```
set OS [hsi::get_os]
common::set_property CONFIG.systmr_dev axi_timer_0 $OS
common::set_property CONFIG.stdin axi_uartlite_0 $OS
common::set_property CONFIG.stdout axi_uartlite_0 $OS
```

#Add custom library to software design

```
hsi::add_library xilflash
```

#Get all the properties of the library, only read_only = false properties can be changed

```
common::report_property [hsi::get_libs xilflash]
```

#Change the default configuration of the library

```
set lib [hsi::get_libs xilflash]
common::set_property CONFIG.enable_amd true $lib
common::set_property CONFIG.enable_intel false $lib
```

```
#Generate the BSP with the above configuration
```

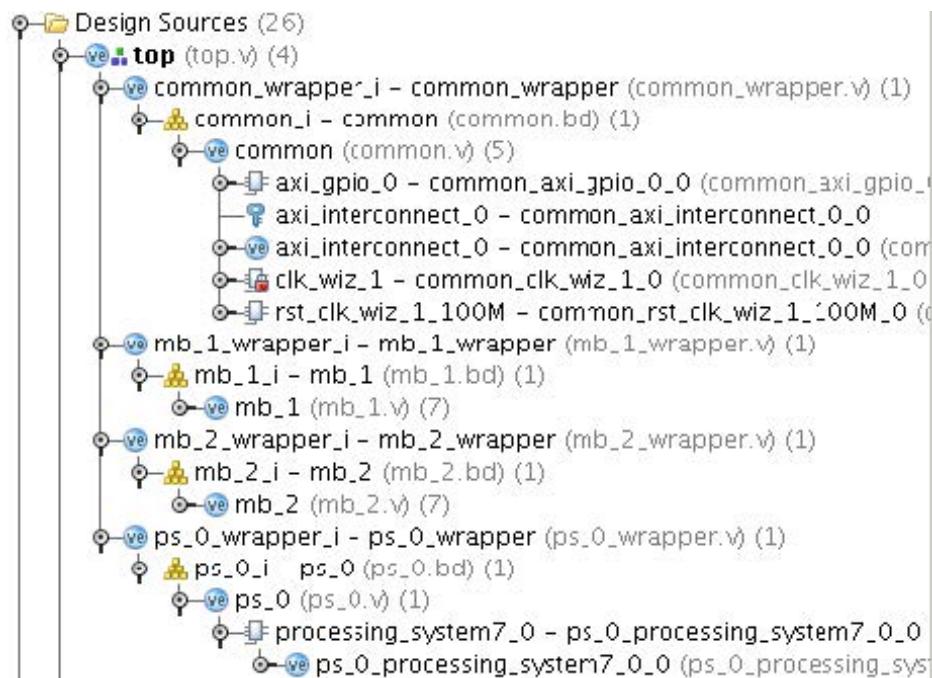
```
hsi::generate_bsp -dir advanced_bsp -compile
```

```
#Delete the library added to software design
```

```
hsi::delete_objs $lib
```

Generating and Compiling BSP for a Multi-Block Design

Figure 71: Example Design with Multiple Block Design Instances in the Active Top Design



```
#Open hardware design with multiple block design instances
```

```
hsim% hsi::open_hw_design system_wrapper.xsa
design_1_wrapper
```

```
#Get the hardware cell instances
```

Note: Cell instances from all the block designs in the top are shown and their names are prefixed with their hierarchy

```
hsim% join [get_cells ] \n
ps_0_wrapper_i-ps_0_i-processing_system7_0
ps7_uart_1
ps7_qspi_0
ps7_cortexa9_0
ps7_cortexa9_1
ps7_ddr_0
```

```
ps7_ethernet_0
...
mb_2_wrapper_i_mb_2_i_axi_gpio_0
mb_2_wrapper_i_mb_2_i_mdm_1
mb_2_wrapper_i_mb_2_i_microblaze_0
mb_2_wrapper_i_mb_2_i_microblaze_0_axi_periph
mb_2_wrapper_i_mb_2_i_microblaze_0_local_memory_dlmb_bram_if_cntlr
mb_2_wrapper_i_mb_2_i_microblaze_0_local_memory_dlmb_v10
mb_2_wrapper_i_mb_2_i_microblaze_0_local_memory_ilmb_bram_if_cntlr
mb_2_wrapper_i_mb_2_i_microblaze_0_local_memory_ilmb_v10
mb_2_wrapper_i_mb_2_i_microblaze_0_local_memory_lmb_bram
mb_2_wrapper_i_mb_2_i_RST_CLK_WIZ_1_100M
mb_1_wrapper_i_mb_1_i_axi_gpio_0
mb_1_wrapper_i_mb_1_i_mdm_1
mb_1_wrapper_i_mb_1_i_microblaze_0
mb_1_wrapper_i_mb_1_i_microblaze_0_axi_periph
mb_1_wrapper_i_mb_1_i_microblaze_0_local_memory_dlmb_bram_if_cntlr
mb_1_wrapper_i_mb_1_i_microblaze_0_local_memory_dlmb_v10
mb_1_wrapper_i_mb_1_i_microblaze_0_local_memory_ilmb_bram_if_cntlr
mb_1_wrapper_i_mb_1_i_microblaze_0_local_memory_ilmb_v10
mb_1_wrapper_i_mb_1_i_microblaze_0_local_memory_lmb_bram
mb_1_wrapper_i_mb_1_i_RST_CLK_WIZ_1_100M
common_wrapper_i_common_i_axi_gpio_0
common_wrapper_i_common_i_axi_interconnect_0
common_wrapper_i_common_i_clk_wiz_1
common_wrapper_i_common_i_RST_CLK_WIZ_1_100M
```

#Generate BSP for a processor in bsp_out directory and compile the bsp sources

```
hsim:generate_bsp -proc mb_2_wrapper_i_mb_2_i_microblaze_0 -dir bsp_out -
compile
ls ./bsp_out(mb_2_wrapper_i_mb_2_i_microblaze_0
code
indent
lib
libsrc
```

HSI Input and Output Files and Specifications

Input Files

XSA

AMD Support Archive (.xsa) is an AMD proprietary file format and only AMD software tools understand it. Third-party software tools can communicate to the XSCT Tcl interface to extract data from the .xsa file.

Note: AMD does not recommend manually editing the XSA file or altering its contents.

XSA is a container and contains:

- One or more .hwh files
 - AMD Vivado™ tool version, part, and board tag information
 - IP - instance, name, VLVN, and parameters

- Memory Map information of the processors
- Internal Connectivity information (including interrupts, clocks, etc.) and external ports information
- BMM/MMI and BIT files
- User and HLS driver files
- Other meta-data files

Software Repository

Default Repositories

By default, the tool scans the following repositories for software components:

- <install>/data/embeddedsw/lib/XilinxProcessorIPLib
- <install>/data/embeddedsw/lib
- <install>/data/embeddedsw/ThirdParty

GIT Repositories

The Device Tree repository can be cloned from Xilinx GIT. Use the `set_repo_path` Tcl command to specify the cloned GIT repository.

User Repositories

You can create drivers, BSPs, and Apps in an example directory structure format, as illustrated in the figure above. Use the `set_repo_path` Tcl command to specify the user repository.

Search Priority Mechanism

The tool uses a search priority mechanism to locate drivers and libraries, as follows:

1. Search the repositories under the library path directory specified using the `set_repo_path` Tcl command.
2. Search the default repositories described above.

Output Files

The tool generates directories, files, and the software design file (MSS) in the <your_project> directory. For every processor instance in the MSS file, the tool generates a directory with the name of the processor instance. Within each processor instance directory the tool generates the following directories and files.

- **The include Directory:** The include directory contains C header files needed by drivers. The include file xparameters.h is also created using the tool in this directory. This file defines base addresses of the peripherals in the system, #defines needed by drivers, OSs, libraries, and user programs, as well as function prototypes.
 - The Microprocessor Driver Definition (MDD) file for each driver specifies the definitions that must be customized for each peripheral that uses the driver. See Microprocessor Driver Definition (MDD) Overview.
 - The Microprocessor Library Definition (MLD) file for each OS and library specifies the definitions that you must customize. See Microprocessor Library Definition (MLD) Overview.
- **The lib Directory:** The lib directory contains `libc.a`, `libm.a`, and `libxil.a` libraries. The libxil library contains driver functions that the particular processor can access. For more information about the libraries, refer to the introductory section of the *BSP and Libraries Document Collection* ([UG643](#)).
- **The libsrc Directory:** The libsrc directory contains intermediate files and make files needed to compile the OSs, libraries, and drivers. The directory contains peripheral-specific driver files, BSP files for the OS, and library files that are copied from install, as well as your driver, OS, and library directories.
- **The code Directory:** The code directory is a repository for tool executables. The tool creates an `xmdstub.elf` file (for the MicroBlaze™ processor on-board debug) in this directory.

Note: The tool removes these directories every time you run the it. You must put your sources, executables, and any other files in an area that you create.

Generating Libraries and Drivers

This section provides an overview of generating libraries and drivers. The hardware specification file and the MSS files define a system. For each processor in the system, the tool finds the list of addressable peripherals. For each processor, a unique list of drivers and libraries are built. The tool does the following for each processor:

- Builds the directory structure, as defined in [Output Files](#).
- Copies the necessary source files for the drivers, OSs, and libraries into the processor instance specific area: `OUTPUT_DIR/processor_instance_name/libsrc`.
- Calls the Design Rule Check (DRC) procedure, which is defined as an option in the MDD or MLD file, for each of the drivers, OSs, and libraries visible to the processor.
- Calls the generate Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor. This generates the necessary configuration files for each of the drivers, OSs, and libraries in the include directory of the processor.
- Calls the `post_generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor.

- Runs make (with targets include and libs) for the OSs, drivers, and libraries specific to the processor. On the Linux platform, the gmake utility is used, while on NT platforms, make is used for compilation.
- Calls the `execs_generate` Tcl procedure (if defined in the Tcl file associated with an MDD or MLD file) for each of the drivers, OSs, and libraries visible to the processor.

MDD, MLD, and Tcl

A driver or library has two associated data files:

- **Data Definition File (MDD or MLD file):** This file defines the configurable parameters for the driver, OS, or library.
- **Data Generation File (Tcl):** This file uses the parameters configured in the MSS file for a driver, OS, or library to generate data. Data generated includes but is not limited to generation of header files, C files, running DRCs for the driver, OS, or library, and generating executables.

The Tcl file includes procedures that tool calls at various stages of its execution. Various procedures in a Tcl file include:

- **DRC:** The name of DRC given in the MDD or MLD file.
- **generate:** A tool-defined procedure that is called after files are copied.
- **post_generate:** A tool-defined procedure that is called after generate has been called on all drivers, OSs, and libraries.
- **execs_generate:** A tool-defined procedure that is called after the BSPs, libraries, and drivers have been generated.

Note: The data generation (Tcl) file is not necessary for a driver, OS, or library.

For more information about the Tcl procedures and MDD/MLD related parameters, refer to [Microprocessor Driver Definition \(MDD\)](#) and [Microprocessor Library Definition \(MLD\)](#).

MSS Parameters

For a complete description of the MSS format and all the parameters that MSS supports, refer to [MSS Overview](#).

Drivers

Most peripherals require software drivers. The peripherals are shipped with associated drivers, libraries and BSPs. Refer to the Device Driver Programmer Guide for more information on driver functions. This guide can be found in the `<install_directory>\vitis \<version>\data\embeddedsw\doc`.

The MSS file includes a driver block for each peripheral instance. The block contains a reference to the driver by name (DRIVER_NAME parameter) and the driver version (DRIVER_VER). There is no default value for these parameters.

A driver has an associated MDD file and a Tcl file.

- The driver MDD file is the data definition file and specifies all configurable parameters for the drivers.
- Each MDD file has a corresponding Tcl file which generates data that includes generation of header files, generation of C files, running DRCs for the driver, and generating executables.

You can write your own drivers. These drivers must be in a specific directory under / or / drivers, as shown in the figure in Software Repository.

- The DRIVER_NAME attribute allows you to specify any name for your drivers, which is also the name of the driver directory.
- The source files and make file for the driver must be in the /src subdirectory under the / directory.
- The make file must have the targets /include and /libs.
- Each driver must also contain an MDD file and a Tcl file in the /data subdirectory.

Open the existing driver files to get an understanding of the required structure.

Refer to [Microprocessor Driver Definition \(MDD\)](#) for details on how to write an MDD and its corresponding Tcl file.

Libraries

The MSS file includes a library block for each library. The library block contains a reference to the library name (LIBRARY_NAME parameter) and the library version (LIBRARY_VER). There is no default value for these parameters. Each library is associated with a processor instance specified using the PROCESSOR_INSTANCE parameter. The library directory contains C source and header files and a make file for the library.

The MLD file for each library specifies all configurable options for the libraries and each MLD file has a corresponding Tcl file.

You can write your own libraries. These libraries must be in a specific directory under /sw_services as shown in the figure in Software Repository.

- The LIBRARY_NAME attribute lets you specify any name for your libraries, which is also the name of the library directory.
- The source files and make file for the library must be in the /src subdirectory under the / directory.
- The make file must have the targets /include and /libs.

- Each library must also contain an MLD file and a Tcl file in the /data subdirectory.

Refer to the existing libraries for more information about the structure of the libraries.

Refer to [Microprocessor Library Definition \(MLD\)](#) for details on how to write an MLD and its corresponding Tcl file.

OS Block

The MSS file includes an OS block for each processor instance. The OS block contains a reference to the OS name (OS_NAME parameter), and the OS version (OS_VER). There is no default value for these parameters. The BSP directory contains C source and header files and a make file for the OS.

The MLD file for each OS specifies all configurable options for the OS. Each MLD file has a corresponding Tcl file associated with it. Refer to [Microprocessor Library Definition \(MLD\)](#) and [Microprocessor Software Specification \(MSS\)](#).

You can write your own OSs. These OSs must be in a specific directory under /bsp, as shown in the figure in Software Repository.

- The OS_NAME attribute allows you to specify any name for your OS, which is also the name of the OS directory.
- The source files and make file for the OS must be in the src subdirectory under the / directory.
- The make file should have the targets /include and /libs.
- Each OS must contain an MLD file and a Tcl file in the /data subdirectory.

Look at the existing OSs to understand the structures. See [Microprocessor Library Definition \(MLD\)](#) Overview for details on how to write an MLD and its corresponding Tcl file, refer to the Device Driver Programmer Guide. This guide is located in your Vitis software platform installation in <install_directory>\vitis\<version> \data\embeddedsw\doc.

Microprocessor Software Specification (MSS)

MSS Overview

The MSS file contains directives for customizing operating systems (OSs), libraries, and drivers.

MSS Format

An MSS file is case insensitive and any reference to a file name or instance name in the MSS file is also case sensitive. Comments can be specified anywhere in the file. A pound (#) character denotes the beginning of a comment, and all characters after it, right up to the end of the line, are ignored. All white spaces are also ignored and carriage returns act as sentence delimiters.

The keywords that are used in an MSS file are as follows:

- **BEGIN:** The keyword begins a driver, processor, or file system definition block. BEGIN should be followed by the driver, processor, or filesys keywords.
- **END:** This keyword signifies the end of a definition block.
- **PARAMETER:** The MSS file has a simple name = value format for statements. The PARAMETER keyword is required before NAME and VALUE pairs. The format for assigning a value to a parameter is parameter name = value. If the parameter is within a BEGIN-END block, it is a local assignment; otherwise it is a global (system level) assignment.

Requirements:

The syntax of various files that the embedded development tools use is described by the Platform Specification Format (PSF). The current PSF version is 2.1.0. The MSS file should also contain version information in the form of parameter Version = 2.1.0, which represents the PSF version 2.1.0.

MSS Example:

An example MSS file follows:

```
parameter VERSION = 2.1.0
BEGIN OS
parameter PROC_INSTANCE = my_microblaze
parameter OS_NAME = standalone
parameter OS_VER = 1.0
parameter STDIN = my_uartlite_1
parameter STDOUT = my_uartlite_1
END
BEGIN PROCESSOR
parameter HW_INSTANCE = my_microblaze
parameter DRIVER_NAME = cpu
parameter DRIVER_VER = 1.0
parameter XMDSTUB_PERIPHERAL = my_jtag
END
BEGIN DRIVER
parameter HW_INSTANCE = my_intc
parameter DRIVER_NAME = intc
parameter DRIVER_VER = 1.0
END
BEGIN DRIVER
parameter HW_INSTANCE = my_uartlite_1
parameter DRIVER_VER = 1.0
parameter DRIVER_NAME = uartlite
END
BEGIN DRIVER
```

```
parameter HW_INSTANCE = my_uartlite_2
parameter DRIVER_VER = 1.0
parameter DRIVER_NAME = uartlite
END
BEGIN DRIVER
parameter HW_INSTANCE = my_timebase_wdt
parameter DRIVER_VER = 1.0
parameter DRIVER_NAME = timebase_wdt
END
BEGIN LIBRARY
parameter LIBRARY_NAME = XilMfs
parameter LIBRARY_VER = 1.0
parameter NUMBYTES = 100000
parameter BASE_ADDRESS = 0x80f00000
END
```

Global Parameters

These parameters are system-specific parameters and do not relate to a particular driver, file system, or library.

PSF Version

This option specifies the PSF version of the MSS file. This option is mandatory, and is formatted as:

```
parameter VERSION = 2.1.0
```

Instance-Specific Parameters

OS, Driver, Library, and Processor Block Parameters

The following list shows the parameters that can be used in OS, driver, library, and processor blocks.

PROC_INSTANCE

This option is required for the OS associated with a processor instances specified in the hardware database, and is formatted as:

```
parameter PROC_INSTANCE = <instance_name>
```

All operating systems require processor instances to be associated with them. The instance name that is given must match the name specified in the hardware database.

HW_INSTANCE

This option is required for drivers associated with peripheral instances specified in the hardware database and is formatted as:

```
parameter HW_INSTANCE = <instance_name>
```

All drivers in software require instances to be associated with the drivers. Even a processor definition block should refer to the processor instance. The instance name that is given must match the name specified in the BD file.

OS_NAME

This option is needed for processor instances that have OSs associated with them and is formatted as:

```
parameter OS_NAME = standalone
```

OS_VER

The OS version is set using the OSVER option and is formatted as:

```
parameter OS_VER = 1.0
```

This version is specified as x.y, where x and y are digits. This is translated to the OS directory searched as follows:

```
OS_NAME_vx_y
```

The MLD (Microprocessor Library Definition) files needed for each OS should be named OS_NAME.mld and should be present in a subdirectory data/ within the driver directory. Refer to [Microprocessor Library Definition \(MLD\)](#) for more information.

DRIVER_NAME

This option is needed for peripherals that have drivers associated with them and is formatted as:

```
parameter DRIVER_NAME = uartlite
```

Library Generator copies the driver directory specified to the OUTPUT_DIR/processor_instance_name/libsrv directory and compiles the drivers using makefiles provided.

DRIVER_VER

The driver version is set using the DRIVER_VER option, and is formatted as:

```
parameter DRIVER_VER = 1.0
```

This version is specified as x.y, where x and y are digits. This is translated to the driver directory searched as follows:

```
DRIVER_NAME_vx_y
```

The MDD (Microprocessor Driver Definition) files needed for each driver should be named `DRIVER_NAME_v2_1_0.mdd` and should be present in a subdirectory `data/` within the driver directory. Refer to [Microprocessor Driver Definition \(MDD\)](#) for more information.

LIBRARY_NAME

This option is needed for libraries, and is formatted as:

```
parameter LIBRARY_NAME = xilmfs
```

The tool copies the library directory specified in the `OUTPUT_DIR/processor_instance_name/libsrv` directory and compiles the libraries using makefiles provided.

LIBRARY_VER

The library version is set using the `LIBRARY_VER` option and is formatted as:

```
parameter LIBRARY_VER = 1.0
```

This version is specified as x.y, where x and y are digits. This is translated to the library directory searched by the tool as follows:

```
LIBRARY_NAME_vx_y
```

The MLD (Microprocessor Library Definition) files needed for each library should be named `LIBRARY_NAME.mld` and should be present in a subdirectory `data/` within the library directory. Refer to [Microprocessor Library Definition \(MLD\)](#) for more information.

MLD/MDD Specific Parameters

Parameters specified in the MDD/MLD file can be overwritten in the MSS file and formatted as:

```
parameter PARAM_NAME = PARAM_VALUE
```

See [Microprocessor Library Definition \(MLD\)](#) and [Microprocessor Driver Definition \(MDD\)](#) for more information.

OS-Specific Parameters

The following list identifies all the parameters that can be specified only in an OS definition block.

STDIN

Identify the standard input device with the STDIN option, which is formatted as:

```
parameter STDIN = instance_name
```

STDOUT

Identify the standard output device with the STDOUT option, which is formatted as:

```
parameter STDOUT = instance_name
```

Example: MSS Snippet Showing OS options

```
BEGIN OS
parameter PROC_INSTANCE = my_microblaze
parameter OS_NAME = standalone
parameter OS_VER = 1.0
parameter STDIN = my_uartlite_1
parameter STDOUT = my_uartlite_1
END
```

Processor-Specific Parameters

Following is a list of all of the parameters that can be specified only in a processor definition block.

XMDSTUB_PERIPHERAL

The peripheral that is used to handle the XMDStub should be specified in the XMDSTUB_PERIPHERAL option. This is useful for the MicroBlaze™ processor only, and is formatted as follows:

```
parameter XMDSTUB_PERIPHERAL = instance_name
```

COMPILER

This option specifies the compiler used for compiling drivers and libraries. The compiler defaults to `mb-gcc` or `powerpc-eabi-gcc` depending on whether the drivers are part of the MicroBlaze processor or PowerPC processor instance. Any other compatible compiler can be specified as an option, and should be formatted as follows:

This example denotes the Diab compiler as the compiler to be used for drivers and libraries.

ARCHIVER

This option specifies the utility to be used for archiving object files into libraries. The archiver defaults to `mb-ar` or `powerpc-eabi-ar` depending on whether or not the drivers are part of the MicroBlaze or PowerPC processor instance. Any other compatible archiver can be specified as an option, and should be formatted as follows:

```
parameter ARCHIVER = ar
parameter COMPILER = dcc
```

This example denotes the archiver `ar` to be used for drivers and libraries.

COMPILER_FLAGS

This option specifies compiler flags to be used for compiling drivers and libraries. If the option is not specified, the tool automatically uses platform and processor-specific options. This option should not be specified in the MSS file if the standard compilers and archivers are used.

The `COMPILER_FLAGS` option can be defined in the MSS if there is a need for custom compiler flags that override generated flags. The `EXTRA_COMPILER_FLAGS` option is recommended if compiler flags must be appended to the ones already generated.

Format this option as follows:

```
parameter COMPILER_FLAGS = ""
```

EXTRA_COMPILER_FLAGS

This option can be used whenever custom compiler flags need to be used in addition to the automatically generated compiler flags, and should be formatted as follows:

```
parameter EXTRA_COMPILER_FLAGS = -g
```

This example specifies that the drivers and libraries must be compiled with debugging symbols in addition to the generated `COMPILER_FLAGS`.

Example: MSS Snippet Showing Processor Options

```
BEGIN PROCESSOR
parameter HW_INSTANCE = my_microblaze
parameter DRIVER_NAME = cpu
parameter DRIVER_VER = 1.00.a
parameter DEFAULT_INIT = xmdstub
parameter XMDSTUB_PERIPHERAL = my_jtag
parameter STDIN = my_uartlite_1
parameter STDOUT = my_uartlite_1
parameter COMPILER = mb-gcc
parameter ARCHIVER = mb-ar
parameter EXTRA_COMPILER_FLAGS = -g -O0
parameter OS = standalone
END
```

Microprocessor Library Definition (MLD)

Microprocessor Library Definition Overview

This section describes the Microprocessor Library Definition (MLD) format, Platform Specification Format 2.1.0. An MLD file contains directives for customizing software libraries and generating Board Support Packages (BSP) for Operating Systems (OS). This document describes the MLD format and the parameters that can be used to customize libraries and OSs.

Requirements

Each OS and library has an MLD file and a Tcl (Tool Command Language) file associated with it. The MLD file is used by the Tcl file to customize the OS or library, depending on different options in the MSS file. For more information on the MSS file format, see [Microprocessor Software Specification \(MSS\)](#). The OS and library source files and the MLD file for each OS and library must be located at specific directories to find the files and libraries.

MLD Library Definition Files

Library Definition involves defining Data Definition (MLD) and a Data Generation (Tcl) files.

Data Definition File

The MLD file (named as `<library_name>.mld` or `<os_name>.mld`) contains the configurable parameters. A detailed description of the various parameters and the MLD format is described in [MLD Parameter Descriptions](#).

Data Generation File

The second file (named as `<library_name>.tcl` or `<os_name>.tcl`, with the filename being the same as the MLD filename) uses the parameters configured in the MSS file for the OS or library to generate data. Data generated includes, but is not limited to, header files, C files, DRCs for the OS or library, and executables. The Tcl file includes procedures that are called by the tool at various stages of its execution. Various procedures in a Tcl file include the following:

- `DRC` (the name of the DRC given in the MLD file)
- `generate` (tool defined procedure) called after OS and library files are copied
- `post_generate` (tool defined procedure) called after generate has been called on all OSs, drivers, and libraries
- `execs_generate` (a tool-defined procedure) called after the BSPs, libraries, and drivers have been generated

Note: An OS/library does not require a data generation file (Tcl file).

MLD Format Specification

The MLD format specification involves the MLD file format specification and the Tcl file format specification. The following subsections describe the MLD.

MLD File Format Specification

The MLD file format specification involves the description of configurable parameters in an OS or a library. The format used to describe this section is discussed in [MLD Parameter Descriptions](#).

Tcl File Format Specification

Each OS and library has a Tcl file associated with the MLD file. This Tcl file has the following sections:

- **DRC:** Contains Tcl routines that validate your OS and library parameters for consistency.
- **Generation:** Contains Tcl routines that generate the configuration header and C files based on the library parameters.

MLD Design Rule Check Section

```
proc mydrc { handle } { }
```

The DRC function could be any Tcl code that checks your parameters for correctness. The DRC procedures can access (read-only) the Platform Specification Format database (which the tool builds using the hardware (XSA) and software (MSS) database files) to read the parameter values that you set. The handle is associated with the current library in the database. The DRC procedure can get the OS and library parameters from this handle. It can also get any other parameter from the database by first requesting a handle and using the handle to get the parameters.

For errors, DRC procedures call the Tcl error command error "error msg" that displays in an error page.

For warnings, DRC procedures return a string value that can be printed on the console.

On success, DRC procedures return without any value.

MLD Format Examples

This section explains the MLD format through an example MLD file and its corresponding Tcl file.

Example: MLD File for a Library

Following is an example of an MLD file for the xilmfs library.

```
option psf_version = 2.1.0 ;
```

option is a keyword identified by the tool. The option name following the option keyword is a directive to the tool to do a specific action.

The psf_version of the MLD file is defined to be 2.1 in this example. This is the only option that can occur before a BEGIN LIBRARY construct now.

```
BEGIN LIBRARY xilmfs
```

The BEGIN LIBRARY construct defines the start of a library named xilmfs.

```
option DESC = "Xilinx Memory File System" ;
option drc = mfs_drc ;
option copyfiles = all;
option REQUIRES_OS = (standalone xilkernel freertos_zynq);
option VERSION = 2.0;
option NAME = xilmfs;
```

The NAME option indicates the name of the driver. The VERSION option indicates the version of the driver.

The COPYFILES option indicates the files to be copied for the library. The DRC option specifies the name of the Tcl procedure that the tool invokes while processing this library. The mfs_drc is the Tcl procedure in the xilmfs.tcl file that would be invoked while processing the xilmfs library.

```
PARAM name = numbytes, desc = "Number of Bytes", type = int, default =
100000, drc = drc_numbytes ;
PARAM name = base_address, desc = "Base Address", type = int, default =
0x10000, drc = drc_base_address ;
PARAM name = init_type, desc = "Init Type", type = enum, values =
("New
file system"=MFSINIT_NEW,
"MFS Image"=MFSINIT_IMAGE, "ROM Image"=MFSINIT_ROM_IMAGE), default =
MFSINIT_NEW ;
PARAM name = need_utils, desc = "Need additional Utilities?", type =
bool, default = false ;
```

PARAM defines a library parameter that can be configured. Each PARAM has the following properties associated with it, whose meanings are self-explanatory: NAME, DESC, TYPE, DEFAULT, RANGE, and DRC. The property VALUES defines the list of possible values associated with an ENUM type.

```
BEGIN INTERFACE file
PROPERTY HEADER="xilmfs.h" ;
FUNCTION NAME=open, VALUE=mfs_file_open ;
FUNCTION NAME=close, VALUE=mfs_file_close ;
FUNCTION NAME=read, VALUE=mfs_file_read ;
FUNCTION NAME=write, VALUE=mfs_file_write ;
FUNCTION NAME=lseek, VALUE=mfs_file_lseek ;
END INTERFACE
```

An interface contains a list of standard functions. A library defining an interface should have values for the list of standard functions. It must also specify a header file where all the function prototypes are defined.

PROPERTY defines the properties associated with the construct defined in the BEGIN construct. Here HEADER is a property with value `xilmfs.h`, defined by the file interface. FUNCTION defines a function supported by the interface.

The `open`, `close`, `read`, `write`, and `lseek` functions of the file interface have the values `mfs_file_open`, `mfs_file_close`, `mfs_file_read`, `mfs_file_write`, and `mfs_file_lseek`. These functions are defined in the header file `xilmfs.h`.

```
BEGIN INTERFACE filesystem
```

BEGIN INTERFACE defines an interface the library supports. Here, `file` is the name of the interface.

```
PROPERTY HEADER= "xilmfs.h" ;
FUNCTION NAME=cd, VALUE=mfs_change_dir ;
FUNCTION NAME=opendir, VALUE=mfs_dir_open ;
FUNCTION NAME=closedir, VALUE=mfs_dir_close ;
FUNCTION NAME=readdir, VALUE=mfs_dir_read ;
FUNCTION NAME=deletedir, VALUE=mfs_delete_dir ;
FUNCTION NAME=pwd, VALUE=mfs_get_current_dir_name ;
FUNCTION NAME=rename, VALUE=mfs_rename_file ;
FUNCTION NAME=exists, VALUE=mfs_exists_file ;
FUNCTION NAME=delete, VALUE=mfs_delete_file ;
END INTERFACE
END LIBRARY
```

END is used with the construct name that was used in the BEGIN statement. Here, END is used with INTERFACE and LIBRARY constructs to indicate the end of each of INTERFACE and LIBRARY constructs.

Example: Tcl File of a Library

The following is the `xilmfs.tcl` file corresponding the `xilmfs.mld` file described in the previous section. The `mfs_drc` procedure would be invoked for the `xilmfs` library while running DRCs for libraries. The generate routine generates constants in a header file and a c file for the `xilmfs` library based on the library definition segment in the MSS file.

```
proc mfs_drc {lib_handle} {
puts "MFS DRC ..."
}
proc mfs_open_include_file {file_name} {
set filename [file join "../../include/" $file_name]
if {[file exists $filename]} {
set config_inc [open $filename a]
} else {
set config_inc [open $filename a]
::hsi::utils::write_c_header $config_inc "MFS Parameters"
}
return $config_inc
```

```
}

proc generate {lib_handle} {
puts "MFS generate ..."
file copy "src/xilmfs.h" "../../include/xilmfs.h"
set conffile [mfs_open_include_file "mfs_config.h"]
puts $conffile "#ifndef _MFS_CONFIG_H"
puts $conffile "#define _MFS_CONFIG_H"
set need_utils [common::get_property CONFIG.need_utils $lib_handle]
if {$need_utils} {
# tell libgen or xps that the hardware platform needs to provide
stdio functions
# inbyte and outbyte to support utils
puts $conffile "#include <stdio.h>"
}
puts $conffile "#include <xilmfs.h>"
set value [common::get_property CONFIG.numbytes $lib_handle]
puts $conffile "#define MFS_NUMBYTES $value"
set value [common::get_property CONFIG.base_address $lib_handle]
puts $conffile "#define MFS_BASE_ADDRESS $value"
set value [common::get_property CONFIG.init_type $lib_handle]
puts $conffile "#define MFS_INIT_TYPE $value"
puts $conffile "#endif"
close $conffile
}
```

Example: MLD File for an OS

An example of an MLD file for the standalone OS is given below:

```
option psf_version = 2.1.0 ;
```

option is a keyword identified by the tool. The option name following the **option** keyword is a directive to the tool to do a specific action. Here the **psf_version** of the MLD file is defined to be 2.1. This is the only option that can occur before a **BEGIN OS** construct at this time.

```
BEGIN OS standalone
```

The **BEGIN OS** construct defines the start of an OS named **standalone**.

```
option DESC = "Generate standalone BSP";
option COPYFILES = all;
```

The **DESC** option gives a description of the MLD. The **COPYFILES** option indicates the files to be copied for the OS.

```
PARAM NAME = stdin, DESC = "stdin peripheral ", TYPE =
peripheral_instance, REQUIRES_INTERFACE = stdin, DEFAULT = none; PARAM
NAME = stdout, DESC = "stdout peripheral ", TYPE = peripheral_instance,
REQUIRES_INTERFACE = stdout, DEFAULT = none ; PARAM NAME = need_xilmalloc,
DESC = "Need xil_malloc?", TYPE = bool, DEFAULT = false ;
```

PARAM defines an OS parameter that can be configured. Each PARAM has the following, associated properties: NAME, DESC, TYPE, DEFAULT, RANGE, DRC. The property VALUES defines the list of possible values associated with an ENUM type.

```
END OS
```

END is used with the construct name that was used in the BEGIN statement. Here END is used with OS to indicate the end of OS construct.

Example: Tcl File of an OS

The following is the `standalone.tcl` file corresponding to the `standalone.mld` file described in the previous section. The generate routine generates constants in a header file and a c file for the `xilmfs` library based on the library definition segment in the MSS file.

```
proc generate {os_handle} {
    global env
    set need_config_file "false"
    # Copy over the right set of files as src based on processor type
    set sw_proc_handle [get_sw_processor]
    set hw_proc_handle [get_cells [get_property HW_INSTANCE
$sw_proc_handle] ]
    set proctype [get_property IP_NAME $hw_proc_handle]
    set procname [get_property NAME $hw_proc_handle]
    set enable_sw_profile [get_property
CONFIG.enable_sw_intrusive_profiling $os_handle]
    set mb_exceptions false
    switch $proctype {
        "microblaze" {
            foreach entry [glob -nocomplain [file join $mbsrcdir *]] {
# Copy over only files that are not related to exception
handling.
# All such files have exception in their names.
file copy -force $entry "./src/"
}
            set need_config_file "true"
            set mb_exceptions [mb_has_exceptions $hw_proc_handle]
}
        "ps7_cortexa9" {
            set procdrv [get_sw_processor]
            set compiler [get_property CONFIG.compiler $procdrv]
            if {[string compare -nocase $compiler "armcc"] == 0} {
set ccdir "./src/cortexa9/armcc"
} else {
set ccdir "./src/cortexa9/gcc"
}
            foreach entry [glob -nocomplain [file join
$cortexa9srcdir *]] {
file copy -force $entry "./src/"
}
            foreach entry [glob -nocomplain [file join $ccdir *]] {
file copy -force $entry "./src/"
}
            file delete -force "./src/armcc"
            file delete -force "./src/gcc"
            if {[string compare -nocase $compiler "armcc"] == 0} {
file delete -force "./src/profile"
            set enable_sw_profile "false"
}
}
    }
}
```

```
set file_handle [xopen_include_file "xparameters.h"]
puts $file_handle "#include \"xparameters_ps.h\""
puts $file_handle ""
close $file_handle
}
"default" {puts "unknown processor type $proctype\n"}
}
```

MLD Parameter Descriptions

MLD Parameter Description Section

This section gives a detailed description of the constructs used in the MLD file.

Conventions

[] Denotes optional values.

< > Value substituted by the MLD writer.

Comments

Comments can be specified anywhere in the file. A “#” character denotes the beginning of a comment and all characters after the “#” right up to the end of the line are ignored. All white spaces are also ignored and semi-colons with carriage returns act as sentence delimiters.

OS or Library Definition

The OS or library section includes the OS or library name, options, dependencies, and other global parameters, using the following syntax:

```
option psf_version = <psf version number> BEGIN LIBRARY/OS <library/os
name> [option drc = <global drc name>] [option depends = <list of
directories>] [option help = <help file>] [option requires_interface =
<list of interface names>] PARAM <parameter description> [BEGIN CATEGORY
<name of category> <category description> END CATEGORY] BEGIN INTERFACE
<interface name> ..... END INTERFACE] END LIBRARY/OS
```

MLD Keywords

The keywords that are used in an MLD file are as follows:

BEGIN

The **BEGIN** keyword begins one of the following: **os**, **library**, **driver**, **block**, **category**, **interface**, and **array**.

END

The `END` keyword signifies the end of a definition block.

PSF_VERSION

Specifies the PSF version of the library.

DRC

Specifies the DRC function name. This is the global DRC function, which is called by the Vitis IDE configuration tool or the command-line tool. This DRC function is called once you enter all the parameters and MLD or MDD writers can verify that a valid OS, library, or driver can be generated with the given parameters.

Option

Specifies that the name following the keyword `option` is an option to the Vitis IDE tools.

OS

Specifies the type of OS. If it is not specified, then OS is assumed as standalone type of OS.

COPYFILES

Specifies the files to be copied for the OS, library, or driver. If `ALL` is used, then the tool copies all the OS, library, or driver files.

DEPENDS

Specifies the list of directories that needs to be compiled before the OS or library is built.

SUPPORTED_PERIPHERALS

Specifies the list of peripherals supported by the OS. The values of this option can be specified as a list, or as a regular expression. For example:

```
option supported_peripherals = (microblaze)
```

Indicates that the OS supports all versions of MicroBlaze. Regular expressions can be used in specifying the peripherals and versions. The regular expression (RE) is constructed as follows:

- Single-Character REs:
 - Any character that is not a special character (to be defined) matches itself.
 - A backslash (followed by any special character) matches the literal character itself. That is, this “escapes” the special character.

- The special characters are: + * ? . [] ^ \$
 - The period (.) matches any character except the new line. For example, .umpty matches both Humpty and Dumpty.
 - A set of characters enclosed in brackets ([]) is a one-character RE that matches any of the characters in that set. For example, [akm] matches either an "a", "k", or "m".
 - A range of characters can be indicated with a dash. For example, [a-z] matches any lowercase letter. However, if the first character of the set is the caret (^), then the RE matches any character except those in the set. It does not match the empty string. Example: [^akm] matches any character except "a", "k", or "m". The caret loses its special meaning if it is not the first character of the set.
- Multi-Character REs:
 - A single-character RE followed by an asterisk (*) matches zero or more occurrences of the RE. Thus, [a-z]* matches zero or more lower-case characters.
 - A single-character RE followed by a plus (+) matches one or more occurrences of the RE. Thus, [a-z]+ matches one or more lower-case characters.
 - A question mark (?) is an optional element. The preceeding RE can occur zero or once in the string, no more. Thus, xy?z matches either xyz or xz.
 - The concatenation of REs is a RE that matches the corresponding concatenation of strings. For example, [A-Z][a-z]* matches any capitalized word.
 - For example, the following matches a version of the axidma:

```
option supported_peripherals = (axi_dma_v[3-9]_[0-9][0-9]_[a-z]
                                 axi_dma_v[3-9]_[0-9]);
```

LIBRARY_STATE

Specifies the state of the library. Following is the list of values that can be assigned to LIBRARY_STATE:

- **ACTIVE:** An active library. By default the value of LIBRARY_STATE is ACTIVE.
- **DEPRECATED:** This library is deprecated
- **OBSOLETE:** This library is obsolete and will not be recognized by any tools. Tools error out on an obsolete library and a new library should be used instead.

APP_COMPILER_FLAGS

This option specifies what compiler flags must be added to the application when using this library. For example:

```
option APP_COMPILER_FLAGS = "-D MYLIBRARY"
```

The Vitis IDE tools can use this option value to automatically set compiler flags automatically for an application.

APP_LINKER_FLAGS

This option specifies that linker flags must be added to the application when using a particular library or OS. For example:

```
option APP_LINKER_FLAGS = "-lxilkernel"
```

The Vitis IDE tools can use this value to set linker flags automatically for an application.

BSP

Specifies a boolean keyword option that can be provided in the MLD file to identify when an OS component is to be treated as a third party BSP. For example:

```
option BSP = true;
```

This indicates that the Vitis tools will offer this OS component as a board support package. If set to false, the component is handled as a native embedded software platform.

OS_STATE

Specifies the state of the operating system (OS). Following is the list of values that can be assigned to OS_STATE:

- **ACTIVE:** This is an active OS. By default the value of OS_STATE is ACTIVE.
- **DEPRECATED:** This OS is deprecated.
- **OBSOLETE:** This OS is obsolete and will not be recognized by the tools. Tools error out on an obsolete OS and a new OS must be specified.
- **OS_TYPE:** Specifies the type of OS. This value is matched with SUPPORTED_OS_TYPES of the driver MDD file for assigning the driver. Default is standalone.
- **REQUIRES_INTERFACE:** Specifies the interfaces that must be provided by other OSs, libraries, or drivers in the system.
- **REQUIRES_OS:** Specifies the list of OSs with which the specified library will work. For example:

```
option REQUIRES_OS = (standalone xilkernel_v4_[0-9][0-9])
```

The Vitis IDE tools use this option value to determine which libraries are offered for a given operating system choice. The values in the list can be regular expressions as shown in the example.

Note: This option must be used on libraries only.

- **HELP:** Specifies the `HELP` file that describes the OS, library, or driver.
- **DEP:** Specifies the condition that must be satisfied before processing an entity. For example to include a parameter that is dependent on another parameter (defined as a DEP, for dependent, condition), the DEP condition should be satisfied. Conditions of the form `(operand1 OP operand2)` are the only supported conditions.
- **INTERFACE:** Specifies the interfaces implemented by this OS, library, or driver. It describes the interface functions and header files used by the library/driver.

```
BEGIN INTERFACE <interface name>
option DEP=;<list of dependencies>;
PROPERTY HEADER=<name of header file where the function is
declared>;
FUNCTION NAME=<name of the interface function>, VALUE=<function
name of library/driver implementation> ;
END INTERFACE
```

- **HEADER:** Specifies the `HEADER` file in which the interface functions would be defined.
- **FUNCTION:** Specifies the `FUNCTION` implemented by the interface. This is a name-value pair in which name is the interface function name and value is the name of the function implemented by the OS, library, or driver.
- **CATEGORY:** Defines an unconditional block. This block gets included based on the default value of the category or if included in the MSS file.

```
BEGIN CATEGORY <category name>
PARAM name = <category name>, DESC=<param description>,
TYPE=<category type>,
DEFAULT=<default>, GUI_PERMIT=<value>, DEP = <condition>
option DEPENDS=<list of dependencies>, DRC=<drc name>, HELP=<help
file>;
<parameters or categories description>
END CATEGORY
```

Nested categories are not supported through the syntax that specifies them. A category is selected in a MSS file by specifying the category name as a parameter with a boolean value `TRUE`. A category must have a `PARAM` with category name.

- **PARAM:** The MLD file has a simple `<name = value>` format for most statements. The `PARAM` keyword is required before every such `NAME, VALUE` pair. The format for assigning a value to a parameter is `param name = <name>, default = value`. The `PARAM` keyword specifies that the parameter can be overwritten in the MSS file.
- **PROPERTY:** Specifies the various properties of the entity defined with a `BEGIN` statement.
- **NAME:** Specifies the name of the entity in which it was defined. (Examples: `param` and `property`.) It also specifies the name of the library if it is specified with `option`.
- **VERSION:** Specifies the version of the library.

- **DESC:** Describes the entity in which it was defined. (Examples: `param` and `property`.)
- **TYPE:** Specifies the type for the entity in which it was defined. (Example: `param`) The following types are supported:
 - **bool:** Boolean (true or false)
 - **int:** integer
 - **string:** String value within " " (quotes)
 - **enum:** List of possible values that a parameter can take
 - **library:** Specify other library that is needed for building the library/driver
 - **peripheral_instance:** Specify other hardware drivers that is needed for building the library
- **DEFAULT:** Specifies the default value for the entity in which it was defined.
- **GUI_PERMIT:** Specifies the permissions for modification of values. The following permissions exist:
 - **NONE:** The value cannot be modified at all.
 - **ADVANCED_USER:** The value can be modified by all. The Vitis IDE does not display this value by default. This is displayed only for the advanced option in the Vitis IDE.
 - **ALL_USERS:** The value can be modified by all. The Vitis IDE displays this value by default. This is the default value for all the values. If `GUI_PERMIT` = `NONE`, the category is always active.
- **ARRAY:** `ARRAY` can have any number of `PARAM`s, and only `PARAM`s. It cannot have `CATEGORY` as one of the fields of an array element. The size of the array can be defined as one of the properties of the array. An array with default values specified in the `default` property leads to its `size` property being initialized to the number of values. If there is no `size` property defined, a `size` property is created before initializing it with the default number of elements. Each parameter in the array can have a default value. In cases in which `size` is defined with an integer value, an array of `size` elements would be created wherein the value of each element would be the default value of each of the parameters.

```
BEGIN ARRAY <array name>
PROPERTY desc = <array description> ;
PROPERTY size = <size of the array>;
PROPERTY default = <List of Values for each element based on the
size of the array>
# array field description as parameters
PARAM name = <name of parameter>, desc = "description of param",
type = <type of param>, default = <default value>
.....
END ARRAY
```

MLD Design Rule Check Section

```
proc mydrc { handle } { }
```

The DRC function could be any Tcl code that checks your parameters for correctness. The DRC procedures can access (read-only) the Platform Specification Format database (which the tool builds using the hardware (XSA) and software (MSS) database files) to read the parameter values that you set. The handle is associated with the current library in the database. The DRC procedure can get the OS and library parameters from this handle. It can also get any other parameter from the database by first requesting a handle and using the handle to get the parameters.

For errors, DRC procedures call the Tcl error command error "error msg" that displays in an error page.

For warnings, DRC procedures return a string value that can be printed on the console.

On success, DRC procedures return without any value.

MLD Tool Generation (Generate) Section

```
proc mygenerate { handle } { }
```

Generate could be any Tcl code that reads your parameters and generates configuration files for the OS or library. The configuration files can be C files, Header files, Makefiles, etc. The generate procedures can access (read-only) the Platform Specification Format database (which the tool builds using the MSS files) to read the parameter values of the OS or library that you set. The handle is a handle to the current OS or library in the database. The generate procedure can get the OS or library parameters from this handle. It can also get any other parameter from the database by first requesting a handle and using the handle to get the parameter.

Microprocessor Driver Definition (MDD)

Microprocessor Driver Definition Overview

A Microprocessor Driver Definition (MDD) file contains directives for customizing software drivers. This document describes the MDD format and the parameters that can be used to customize drivers.

Requirements

Each device driver has an MDD file and a Tool Command Language (Tcl) file associated with it. The MDD file is used by the Tcl file to customize the driver, depending on different options configured in the MSS file. For more information on the MSS file format, see [Microprocessor Software Specification \(MSS\)](#).

The driver source files and the MDD file for each driver must be located at specific directories in order to find the files and the drivers. This document describes the MDD format and the parameters that can be used to customize drivers.

MDD Driver Definition Files

Driver Definition involves defining a Data Definition file (MDD) and a Data Generation file (Tcl file).

- **Data Definition File:** The MDD file (`<driver_name>.mdd`) contains the configurable parameters. A detailed description of the parameters and the MDD format is described in [MDD Parameter Description](#).
- **Data Generation File:** The second file (`<driver_name>.tcl`), with the filename being the same as the MDD filename) uses the parameters configured in the MSS file for the driver to generate data. Data generated includes but is not limited to generation of header files, C files, running DRCs for the driver, and generating executables. The Tcl file includes procedures that are called by the tool at various stages of its execution.

Various procedures in a Tcl file includes: the DRC (name of the DRC given in the MDD file), `generate` (tool defined procedure) called after driver files are copied, `post_generate` (tool defined procedure) called after `generate` is called on all drivers and libraries, and `execs_generate` called after the libraries and drivers are generated.

Note: A driver does not require the data generation file (Tcl file).

MDD Format Specification

The MDD format specification involves the MDD file Format specification and the Tcl file Format specification which are described in the following subsections.

MDD File Format Specification

The MDD file format specification describes the parameters defined in the Parameter Description section. This data section describes configurable parameters in a driver. The format used to describe these parameters is discussed in [MDD Parameter Description](#).

Tcl File Format Specification

Each driver has a Tcl file associated with the MDD file. This Tcl file has the following sections:

- **DRC Section:** This section contains Tcl routines that validate your driver parameters for consistency.
- **Generation Section:** This section contains Tcl routines that generate the configuration header and C files based on the driver parameters.

MDD Format Examples

This section explains the MDD format through an example of an MDD file and its corresponding Tcl file.

Example: MDD File

The following is an example of an MDD file for the uartlite driver.

```
option psf_version = 2.1;
```

option is a keyword identified by the tool. The option name following the option keyword is a directive to the tool to do a specific action. Here the psf_version of the MDD file is defined as 2.1. This is the only option that can occur before a BEGIN DRIVER construct.

```
BEGIN DRIVER uartlite
```

The BEGIN DRIVER construct defines the start of a driver named uartlite.

```
option supported_peripherals = (mdm axi_uartlite);  
option driver_state = ACTIVE;  
option copyfiles = all;  
option VERSION = 3.0;  
option NAME = uartlite;
```

The NAME option indicates the name of the driver. The VERSION option indicates the version of the driver. The COPYFILES option indicates the files to be copied for a “level” 0 uartlite driver.

```
BEGIN INTERFACE stdin
```

BEGIN INTERFACE defines an interface the driver supports. The interface name is stdin.

```
PROPERTY header = xuartlite_l.h;  
FUNCTION name = inbyte, value = XUartLite_RecvByte;  
END INTERFACE
```

An Interface contains a list of standard functions. A driver defining an interface should have values for the list of standard functions. It must also specify a header file in which all the function prototypes are defined.

PROPERTY defines the properties associated with the construct defined in the BEGIN construct. The header is a property with the value `xuartlite_1.h`, defined by the `stdin` interface. FUNCTION defines a function supported by the interface. The `inbyte` function of the `stdin` interface has the value `XUartLite_RecvByte`. This function is defined in the header file `xuartlite_1.h`.

```
BEGIN INTERFACE stdout
PROPERTY header = xuartlite_1.h;
FUNCTION name = outbyte, value = XUartLite_SendByte;
END INTERFACE
BEGIN INTERFACE stdio
PROPERTY header = xuartlite_1.h;
FUNCTION name = inbyte, value = XUartLite_RecvByte;
FUNCTION name = outbyte, value = XUartLite_SendByte;
END INTERFACE
```

END is used with the construct name that was used in the BEGIN statement. Here END is used with BLOCK and DRIVER constructs to indicate the end of each BLOCK and DRIVER construct.

Example: Tcl File

The following is the `uartlite.tcl` file corresponding to the `uartlite.mdd` file described in the previous section. The “`uartlite_drc`” procedure would be invoked for the `uartlite` driver while running DRCs for drivers. The generate routine generates constants in a header file and a `c` file for `uartlite` driver, based on the driver definition segment in the MSS file.

```
proc generate {drv_handle} {
::hsi::utils::define_include_file $drv_handle "xparameters.h"
"XUartLite" "NUM_INSTANCES" "C_BASEADDR"
"C_HIGHADDR" "DEVICE_ID" "C_BAUDRATE" "C_USE_PARITY" "C_ODD_PARITY"
"C_DATA_BITS"
::hsi::utils::define_config_file $drv_handle "xuartlite_g.c"
"XUartLite" "DEVICE_ID" "C_BASEADDR"
"C_BAUDRATE" "C_USE_PARITY" "C_ODD_PARITY" "C_DATA_BITS"
::hsi::utils::define_canonical_xpars $drv_handle "xparameters.h"
"UartLite" "DEVICE_ID" "C_BASEADDR"
"C_HIGHADDR" "C_BAUDRATE" "C_USE_PARITY" "C_ODD_PARITY" "C_DATA_BITS"
}
```

MDD Parameter Description

This section gives a detailed description of the constructs used in the MDD file.

Conventions

[]: Denotes optional values.

< >: Value substituted by the MDD writer.

Comments

Comments can be specified anywhere in the file. A pound (#) character denotes the beginning of a comment, and all characters after it, right up to the end of the line, are ignored. All white spaces are also ignored and semicolons with carriage returns act as sentence delimiters.

Driver Definition

The driver section includes the driver name, options, dependencies, and other global parameters, using the following syntax:

```
option psf_version = <  
psf version number>  
BEGIN DRIVER <driver name>  
[option drc = <global drc name>]  
[option depends = <list of directories>]  
[option help = <help file>]  
[option requires_interface = <list of interface names>  
]  
PARAM <parameter description>  
[BEGIN BLOCK,dep = <condition>  
.....  
END BLOCK]  
[BEGIN INTERFACE <interface name>  
.....  
END INTERFACE]  
END DRIVER
```

MDD Keywords

The keywords that are used in an MDD file are as follows:

Begin

The BEGIN keyword begins with one of the following: library, drive, block, category, or interface.

END

The END keyword signifies the end of a definition block.

PSF_VERSION

Specifies the PSF version of the library.

DRC

Specifies the DRC function name. This is the global DRC function that is called by the Vitis IDE configuration tool or the command-line tool. This DRC function is called when you enter all the parameters and the MLD or MDD writers can verify that a valid library or driver can be generated with the given parameters.

option

Specifies the name following the keyword option is an option to the tool. The following five options are supported: COPYFILES, DEPENDS, SUPPORTED_PERIPHERALS, and DRIVER_STATE.

SUPPORTED_OS_TYPES

Specifies the list of supported OS types. If it is not specified, then driver is assumed as standalone driver.

COPYFILES

Specifies the list of files to be copied for the driver. If ALL is specified as the value, the tool copies all the driver files.

DEPENDS

Specifies the list of directories on which a driver depends for compilation.

SUPPORTED_PERIPHERALS

Specifies the list of peripherals supported by the driver. The values of this option can be specified as a list or as a regular expression. The following example indicates that the driver supports all versions of opb_jtag_uart and the opb_uartlite_v1_00_b version:

```
option supported_peripherals = (xps_uartlite_v1_0, xps_uart16550)
```

Regular expressions can be used in specifying the peripherals and versions. The regular expression (RE) is constructed as described below.

Single-Character REs

- Any character that is not a special character (to be defined) matches itself.
- A backslash (followed by any special character) matches the literal character itself. That is, it escapes the special character.
- The special characters are: + * ? . [] ^ \$
- The period matches any character except the newline. For example, .umpty matches both Humpty and Dumpty.
- A set of characters enclosed in brackets ([]) is a one-character RE that matches any of the characters in that set. For example, [akm] matches an a, k, or m. A range of characters can be indicated with a dash. For example, [a-z] matches any lower-case letter.

However, if the first character of the set is the caret (^), then the RE matches any character except those in the set. It does not match the empty string. For example, [^akm] matches any character except a, k, or m. The caret loses its special meaning if it is not the first character of the set.

Multi-Character REs

- A single-character RE followed by an asterisk (*) matches zero or more occurrences of the RE. Therefore, [a-z]* matches zero or more lower-case characters.
- A single-character RE followed by a plus (+) matches one or more occurrences of the RE. Therefore, [a-z]+ matches one or more lower-case characters.
- A question mark (?) is an optional element. The preceding RE can occur no times or one time in the string. For example, xy?z matches either xyz or xz.
- The concatenation of REs is an RE that matches the corresponding concatenation of strings. For example, [A-Z][a-z]* matches any capitalized word.

The following example matches any version of xps_uartlite, xps_uart16550, and mdm.

```
option supported_peripherals = (xps_uartlite_v[0-9]+_[1-9][0-9]_[a-z]  
xps_uart16550 mdm);
```

DRIVER_STATE

Specifies the state of the driver. The following are the list of values that can be assigned to DRIVER_STATE:

- **ACTIVE:** This is an active driver. By default the value of DRIVER_STATE is ACTIVE.
- **DEPRECATED:** This driver is deprecated and is scheduled to be removed.
- **OBsolete:** This driver is obsolete and is not recognized by any tools. Tools error out on an obsolete driver, and a new driver should be used instead.

REQUIRES_INTERFACE

Specifies the interfaces that must be provided by other libraries or drivers in the system.

HELP

Specifies the help file that describes the library or driver.

DEP

Specifies the condition that needs to be satisfied before processing an entity. For example, to enter into a BLOCK, the DEP condition should be satisfied. Conditions of the form (operand1 OP operand2) are supported.

BLOCK

Specifies the block is to be entered into when the DEP condition is satisfied. Nested blocks are not supported.

INTERFACE

Specifies the interfaces implemented by this library or driver and describes the interface functions and header files used by the library or driver.

```
BEGIN INTERFACE <interface name>
option DEP=<list of dependencies>;
PROPERTY HEADER=<name of header file where the function is declared>
;
FUNCTION NAME=<name of interface function>, VALUE=<function name
of library/driver implementation> ;
END INTERFACE
```

HEADER

Specifies the header file in which the interface functions would be defined.

FUNCTION

Specifies the function implemented by the interface. This is a name-value pair where name is the interface function name and value is the name of the function implemented by the library or driver.

PARAM

Generally, the MLD/MDD file has a name = value format for statements. The PARAM keyword is required before every such NAME, VALUE pair. The format for assigning a value to a parameter is param name = <name>, default= value. The PARAM keyword specifies that the parameter can be overwritten in the MSS file.

DTGPARAM

The DTGPARAM keyword is specially used for the device-tree specific parameters that can be configured. Driver defines these DTGPARAMs if it needs to dump any parameters in the Tool DTG generated DTS file.

PROPERTY

Specifies the various properties of the entity defined with a BEGIN statement.

NAME

Specifies the name of the entity in which it was defined (example: PARAM, PROPERTY). It also specifies the name of the driver if it is specified with option.

VERSION

Specifies the version of the driver.

DESC

Describes the entity in which it was defined (example: PARAM, PROPERTY).

TYPE

Specifies the type for the entity in which it was defined (example: PARAM). The following are the supported types:

- bool: Boolean (true or false)

int: Integer

string: String value within " " (quotes).

enum: List of possible values, that this parameter can take.

library: Specify other library that is needed for building the library or driver.

peripheral_instance: Specify other hardware drivers needed for building the library or driver.

Regular expressions can be used to specify the peripheral instance. Refer to

SUPPORTED_PERIPHERALS in [MLD Keywords](#) for more details about regular expressions.

DEFAULT

Specifies the default value for the entity in which it was defined.

GUI_PERMIT

Specifies the permissions for modification of values. The following permissions exist:

- **NONE**: The value can not be modified at all.
- **ADVANCED_USER**: The value can be modified by all. The Vitis IDE does not display this value by default. It is displayed only as an advanced option in the Vitis IDE.
- **ALL_USERS**: The value can be modified by all. The Vitis IDE displays this value by default. This is the default value for all the values. If GUI_PERMIT = NONE, the category is always active.

MDD Design Rule Check (DRC) Section

```
proc mydrc { handle }
```

The DRC function can be any Tcl code that checks your parameters for correctness. The DRC procedures can access (read-only) the Platform Specification Format database (built by the tool using the hardware (XSA) and software (MSS) database files) to read the parameter values you set. The "handle" is a handle to the current driver in the database. The DRC procedure can get the driver parameters from this handle. It can also get any other parameter from the database by first requesting a handle and then using the handle to get the parameters.

- For errors, DRC procedures call the Tcl error command error "error msg" that displays in an error page.
- For warnings, DRC procedures return a string value that can be printed on the console.
- On success, DRC procedures return without any value.

MDD Driver Generation (Generate) Section

```
proc mygenerate { handle }
```

generate could be any Tcl code that reads your parameters and generates configuration files for the driver. The configuration files can be C files, Header files, or Makefiles. The generate procedures can access (read-only) the Platform Specification Format database (built by the tool using the MSS files) to read the parameter values of the driver that you set. The handle is a handle to the current driver in the database. The generate procedure can get the driver parameters from this handle. It can also get any other parameters from the database by requesting a handle and then using the handle to get the parameter.

Custom Driver

This section demonstrates how to hand-off a custom driver associated with an IP(driver files are specified in IPXACT file of the IP component) and access the driver information in HSI as well as associate the driver with IP during BSP generation. For more information on packaging IP with custom driver, refer to *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#)).

An example design of an IP with custom driver specified in its IPXACT definition.

Figure 72: Example Design with an IP with Custom Driver

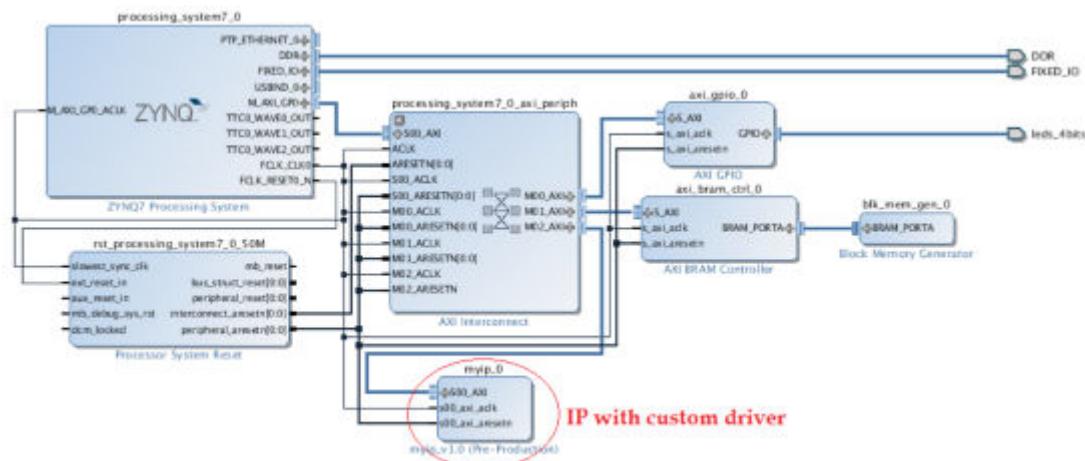


Figure 73: Custom Driver Specified in IPXACT Specification of an IP

```
<spirit:fileSet>
  <spirit:name>xilinx softwaredriver view fileset</spirit:name>
  <spirit:file>
    <spirit:name>drivers/myip_v1_0/data/myip.mdd</spirit:name>
    <spirit:userFileType>mdd</spirit:userFileType>
    <spirit:userFileType>driver_mdd</spirit:userFileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>drivers/myip_v1_0/data/myip.tcl</spirit:name>
    <spirit:fileType>tclSource</spirit:fileType>
    <spirit:userFileType>driver_tcl</spirit:userFileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>drivers/myip_v1_0/src/Makefile</spirit:name>
    <spirit:userFileType>unknown</spirit:userFileType>
    <spirit:userFileType>driver_src</spirit:userFileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>drivers/myip_v1_0/src/myip.h</spirit:name>
    <spirit:fileType>cSource</spirit:fileType>
    <spirit:userFileType>driver_src</spirit:userFileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>drivers/myip_v1_0/src/myip.c</spirit:name>
    <spirit:fileType>cSource</spirit:fileType>
    <spirit:userFileType>driver_src</spirit:userFileType>
  </spirit:file>
  <spirit:file>
    <spirit:name>drivers/myip_v1_0/src/myip_selftest.c</spirit:name>
    <spirit:fileType>cSource</spirit:fileType>
    <spirit:userFileType>driver_src</spirit:userFileType>
  </spirit:file>
</spirit:fileSet>
```

Custom driver specified in IPXACT specification of an IP

Run Vivado hardware hand-off flow either in Pre-Synth or Post-Bitstream mode. The custom driver for each IP is packaged in an XSA.

Open the hardware design with custom drivers.

```
hsi::open_hw_design ./base_zynq_design_wrapper.xsa
  base_zynq_design_wrapper
```

Create a software design

```
hsi::create_sw_design swdesign -proc ps7_cortexa9_0 -os standalone
  Swdesign
```

```
# Check if the custom drivers are assigned to respective IP cores or not
```

```
join [hsi::get_drivers ] \n
      axi_bram_ctrl_0
      axi_gpio_0
      myip_0
```

```
# Check the custom driver properties
```

```
common::report_property [ hsi::get_drivers myip* ]
```

Table 59: Example Table

Property	Type	Read-only	Visible	Value
CLASS	string	true	true	driver
HW_INSTANCE	string	true	true	myip_0
NAME	string	false	true	myip
VERSION	string	false	true	1.0

```
# Generate BSP. BSP source code including custom driver sources will be dumped to the bsp_out
#directory
```

```
hsi::generate_bsp -dir bsp_out
base_zynq_design_wrapper
ls ./bsp_out/ps7_cortexa9_0/libsrc/
.
.
myip_v1_0
.
.
```

Microprocessor Application Definition (MAD)

Microprocessor Application Definition Overview

A MAD file contains directives for customizing software application. This section describes the Microprocessor Application Definition (MAD) format, Platform Specification Format 2.1.0. and the parameters that can be used to customize applications.

Requirements

Each application has an MAD file and a Tool Command Language (Tcl) file associated with it.

The MAD file is used by Hsi to recognize it as an application and to consider its configuration while generating the application sources. The MAD file for each application must be located in its data directory.

Microprocessor Application Definition Files

Application Definition involves defining a Microprocessor Application Definition file (MAD) and a Data Generation file (Tcl file).

Application Definition File

The MAD file (<application_name>.mad) contains the name, description and other configurable parameters. A detailed description of the various parameters and the MAD format is described in [MAD Format Specification](#).

Data Generation File

The second file (<application_name>.tcl, with the filename being the same as the MAD filename) uses the parameters in the MAD file for the application to generate data.

Data generated includes, but is not limited to, generation of header files, C files, running DRCs for the application and generating executables. The Tcl file includes procedures that are called by the tool at various stages of its execution. Various procedures in a Tcl file includes the following:

- DRC (swapp_is_supported_hw, swapp_is_supported_sw)
- swapp_generate (tool defined procedure) called after application source files are copied

MAD Format Specification

The MAD format specification involves the MAD file format specification and the Tcl file format specification.

MAD File Format Specification

The MAD file format specification describes the parameters using a sample MAD file and its corresponding Tcl file.

The following example shows a MAD file for a sample application called my_application.

```
option psf_version = 2.1;
```

option is a keyword identified by the tool. The option name following the option keyword is a directive to the tool to do a specific action.

The psf_version of the MAD file is defined to be 2.1 in this example. This is the only option that can occur before a BEGIN APPLICATION construct .

```
BEGIN APPLICATION my_application
```

The BEGIN APPLICATION construct defines the start of an application named my_application.

```
option NAME = myapplication
        option DESCRIPTION = "My custom application"
END APPLICATION
```

Note: The application NAME should match the return value of the Tcl process `swapp_get_name` in the application Tcl file described above.

Tcl File Format Specification

Each application has a Tcl file associated with the MAD file. This Tcl file has the following sections:

- *DRC Section*: This section contains Tcl routines that validate your hardware and software instances and their configuration needed for the application.
- *Generation Section*: This section contains Tcl routines that generate the application header and C files based on the hardware and software configuration.

MAD Format Example

This section explains the MAD format through an example MAD file and its corresponding Tcl file.

Example: MAD File

The following is an example of an MAD file for a sample application called my_application.

```
option psf_version = 2.1;
```

`option` is a keyword identified by the tool. The option name following the `option` keyword is a directive to the tool to do a specific action.

The `psf_version` of the MAD file is defined to be 2.1 in this example. This is the only option that can occur before a BEGIN APPLICATION construct .

```
BEGIN APPLICATION my_application
```

The BEGIN APPLICATION construct defines the start of an application named my_application.

```
option NAME = myapplication
        option DESCRIPTION = "My custom application"
END APPLICATION
```

Note: Application NAME should match the return value of Tcl proc `swapp_get_name` in application Tcl file described above.

HSI Commands

This section contains all hardware and software interface Tcl commands, arranged alphabetically.

common::get_property

Description

Get properties of object.

Syntax

```
get_property [-min] [-max] [-quiet] [-verbose] <name> <object>
```

Returns

Property value.

Usage

Name	Description
[-min]	Return only the minimum value
[-max]	Return only the maximum value
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
<name>	Name of property whose value is to be retrieved
<object>	Object to query for properties

Categories

Object, PropertyAndParameter

Description

Gets the current value of the named property from the specified object or objects. If multiple objects are specified, a list of values is returned.

If the property is not currently assigned to the object, or is assigned without a value, then the `get_property` command returns nothing, or the null string. If multiple objects are queried, the null string is added to the list of values returned.

This command returns a value, or list of values, or returns an error if it fails.

Arguments

-min - (optional) When multiple objects are specified, this option examines the values of the named property, and returns the smallest value from the list of objects. Numeric properties are sorted by value. All other properties are sorted as strings.

-max - (optional) When multiple objects are specified, this option examines the values of the named property, and returns the largest value from the list of objects. Numeric properties are sorted by value. All other properties are sorted as strings.

-quiet - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

name - (required) The name of the property to be returned. The name is not case sensitive.

object - (required) One or more objects to examine for the specified property.

Examples

Get the `NAME` property from the specified cell:

```
common::get_property NAME [lindex [get_cells] 0]
```

Get the `BOARD` property from the current hardware design:

```
common::get_property BOARD [current_hw_design]
```

common::report_property

Description

Report properties of object.

Syntax

```
report_property [-all] [-class <arg>] [-return_string] [-file <arg>] [-append] [-regexp] [-quiet] [-verbose] [<object>] [<pattern>]
```

Returns

Property report.

Usage

Name	Description
<code>[-all]</code>	Report all properties of object even if not set
<code>[-class]</code>	Object type to query for properties. Not valid with <code><object></code>
<code>[-return_string]</code>	Set the result of running <code>report_property</code> in the Tcl interpreter's result variable
<code>[-file]</code>	Filename to output result to. Send output to console if <code>-file</code> is not used
<code>[-append]</code>	Append the results to file; do not overwrite the results file
<code>[-regexp]</code>	Pattern is treated as a regular expression
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code><object></code>	Object to query for properties
<code><pattern></code>	Pattern to match properties against Default: *

Categories

Object, PropertyAndParameter, Report

Description

Gets the property name, property type, and property value for all of the properties on a specified object, or class of objects.

Note: `list_property` also returns a list of all properties on an object, but does not include the property type or value.

You can specify objects for `report_property` using the `get_*` series of commands to get a specific object. You can use the `lindex` command to return a specific object from a list of objects:

```
report_property [lindex [get_cells] 0]
```

However, if you are looking for the properties on a class of objects, you should use the `-class` option instead of an actual object.

This command returns a report of properties on the object, or returns an error if it fails.

Arguments

`-all>` - (optional) Return all of the properties for an object, even if the property value is not currently defined.

-class <arg>- (optional) Return the properties of the specified class instead of a specific object. The class argument is case sensitive, and most class names are lower case.

Note: -class cannot be used together with an <object>

-return_string- (optional) Directs the output to a Tcl string. The Tcl string can be captured by a variable definition and parsed or otherwise processed.

-file <arg>- (optional) Write the report into the specified file. The specified file will be overwritten if one already exists, unless -append is also specified.

Note: If the path is not specified as part of the file name, the file will be written into the current working directory, or the directory from which the tool was launched.

-append - (optional) Append the output of the command to the specified file rather than overwriting it.

Note: The -append option can only be used with the -file option.

-regexp- (optional) Specifies that the search <pattern> is written as a regular expression.

-quiet - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

<object> - (optional) A single object on which to report properties.

Note: If you specify multiple objects you will get an error.

<pattern> - (optional) Match the available properties on the <object> or -class against the specified search pattern. The <pattern> applies to the property name, and only properties matching the specified pattern will be reported. The default pattern is the wildcard `*` which returns a list of all properties on the specified object.

Note: The search pattern is case sensitive, and most properties are UPPER case.

Examples

The following example returns all properties of the specified object:

```
common::report_property -all [get_cells microblaze_0]
```

To determine which properties are available for the different design objects supported by the tool, you can use multiple `report_property` commands in sequence. The following example returns all properties of the specified current objects:

```
common::report_property -all [current_hw_design]
```

```
common::report_property -all [current_sw_design]
```

hsi::close_hw_design

Description

Close a hardware design.

Syntax

```
close_hw_design [-quiet] [-verbose] <name>
```

Returns

Returns nothing, error message if failed.

Usage

Name	Description
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code><name></code>	Name of design to close

Categories

Hardware

Description

Closes the hardware design in the HSM active session. Design modification is not allowed in the current release, otherwise it will prompt to save the design prior to closing.

Arguments

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

<name> - The name of the hardware design object to close.

Examples

Close the current hardware design object:

```
hsi::close_hw_design [current_hw_design]
```

Close the specified hardware design object:

```
hsi::close_hw_design design_1_imp
```

hsi::create_dt_node

Description

Create a DT node.

Syntax

```
create_dt_node -name <arg> [-unit_addr <arg>] [-label <arg>] [-objects <args>] [-quiet] [-verbose]
```

Returns

DT node object. Returns nothing if the command fails.

Usage

Name	Description
<code>-name</code>	Child DT node name
<code>[-unit_addr]</code>	Unit address of node
<code>[-label]</code>	Label of node
<code>[-objects]</code>	List of nodes
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution

Categories

DeviceTree

Description

Create a new DT node and add to the current DT tree.

If successful, this command returns the name of the DT node created where name is represented as "node_label"+ "node_name" + "@unit_address". Otherwise it returns an error.

Arguments

-name - The name of the node to be created.

-label - The label of the node to represent in generated dtsi file.

--unit_addr - The unit address of the node to represent in generated dtsi file.

-objects - The list of node objects where the newly created node will be a child to all specified nodes.

-quiet - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

Examples

Create a new DT node amba with lable axi_interconnect and unit_addr 0x000 in the current DT tree:

```
hsi::create_dt_node -name amba -label axi_interconnect -unit_addr 0x0000
```

```
hsi::create_dt_node -name amba -label axi_interconnect -unit_addr 0x0000 -  
objects [get_dt_nodes -of_objects\>
```

hsi::create_dt_tree

Description

Create a DT tree.

Syntax

```
create_dt_tree -dts_file <arg> [-dts_version <arg>] [-quiet] [-verbose]
```

Returns

Tree object. Returns nothing if the command fails.

Usage

Name	Description
-dts_file	dts file name
[-dts_version]	dts version
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution

Categories

DeviceTree

Description

Create a new DT tree add to the current HSI session.

If successful, this command returns the name of the DT tree created. Otherwise it returns an error.

Arguments

`-dts_file` - The DT tree name or file name targeted for the output DTSI file.

`-dts_version` - The DTS version of the DTSI file.

`-verbose` - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

Examples

Create a new DT tree `pl.dtsi` and add the tree to the current session:

```
hsic::create_dt_tree -dts_file pl.dtsi -dts_version /dts-v1/
```

```
hsic::create_dt_tree -dts_file system.dts -dts_version /dts-v3/ -  
header "include pl.dtsi, include ps.dtsi"
```

```
hsic::create_dt_tree -dts_file ps.dtsi -dts_version /dts-v3/ -  
header "PS system info"
```

hsi::get_cells

Description

Get a list of cells.

Syntax

```
get_cells [-regexp] [-filter <arg>] [-hierarchical] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Cell objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-hierarchical]	Get cells from all levels of hierarchical cells
[-of_objects]	Get 'cell' objects of these types: 'hw_design port bus_intf net_intf_net'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of IP instance objects in the current design that match a specified search pattern. The default command returns a list of all IP instances in the design.

Note: To improve memory and performance, the commands return a container list of a single type of objects (e.g. cells, nets, or ports). You can add new objects to the list (using lappend for instance), but you can only add the same type of object that is currently in the list. Adding a different type of object, or string, to the list is not permitted and will result in a Tcl error.

Arguments

`-regexp` – (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `. *` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `" "`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For cell objects, "IP_TYPE", and "IP_NAME" are some of the properties you can use to filter results. The following gets cells with an IP_TYPE of "PROCESSOR" and with names containing "ps7":

```
get_cells * -filter {IP_TYPE == PROCESSOR && NAME !~ "*ps7*"}
```

`-hierarchical` – (optional) Get cells from all levels of hierarchical cells .

`-of_objects <arg>` – (optional) Get the cells connected to the specified pins, timing paths, nets, bels, clock regions, sites or DRC violation objects.

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` – (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`<patterns>` - (optional) Match cells against the specified patterns. The default pattern is the wildcard `'*'` which gets a list of all cells in the project. More than one pattern can be specified to find multiple cells based on different search criteria.

Note: You must enclose multiple search patterns in braces, {}, or quotes, "", to present the list as a single element.

Examples

The following example returns list of processor instances :

```
hsi::get_cells -filter { IP_TYPE == "PROCESSOR" }
```

This example gets a list of properties and property values attached to the second object of the list returned by `get_cells`:

```
common::report_property [lindex [get_cells] 1]
```

Note: If there are no cells matching the pattern you will get a warning.

hsi::get_dt_nodes

Description

Get a list of DT node objects.

Syntax

```
get_dt_nodes [-hier] [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Node objects. Returns nothing if the command fails.

Usage

Name	Description
<code>[-hier]</code>	List of nodes in the current tree.
<code>[-regexp]</code>	Patterns are full regular expressions
<code>[-filter]</code>	Filter list with expression
<code>[-of_objects]</code>	Get "" objects of these types: 'dtsNode dtsTree'.
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution

Name	Description
[<patterns>]	Match cell names against patterns Default: *

Categories

DeviceTree

Description

Gets a list of DT nodes created under a DT tree in the current HSI session that match a specified search pattern. The default command gets a list of all root DT nodes in the current DT tree.

Arguments

`-of_objects <arg>` - (optional) Gets all nodes of DTSNode and DTSTree

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_dt_nodes` or `get_dt_trees`, rather than specifying objects by name. In addition, `-of_objects` cannot be used with a search `<pattern>`.

`-regexp` - (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` - (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

`-quiet` - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`<patterns>` - (optional) Match nodes against the specified patterns. The default pattern is the wildcard `'*'` which gets a list of all root nodes in the current DT tree. More than one pattern can be specified to find multiple nodes based on different search criteria.

Note: You must enclose multiple search patterns in braces, {}, or quotes, "", to present the list as a single element.

Examples

The following example gets a list of root nodes attached to the specified DT tree:

```
hsi::get_dt_nodes -of_objects [lindex [get_dt_trees] 1]
```

Note: If there are no nodes matching the pattern, the tool will return empty.

The following example gets a list of all nodes in the current DT tree:

```
hsi::get_dt_nodes -hier
```

Note: If there are no nodes matching the pattern, the tool will return empty.

The following example gets a list of nodes created under a root node:

```
hsi::get_dt_nodes -of_objects [current_dt_tree]
```

Note: If there are no nodes matching the pattern, the tool will return empty.

hsi::get_dt_trees

Description

Get a list of dts trees created.

Syntax

```
get_dt_trees [-regexp] [-filter <arg>] [-quiet] [-verbose] [<patterns>...]
```

Returns

DTS tree objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match tree names against patterns Default: *

Categories

DeviceTree

Description

Gets a list of DT trees created in the current HSI session that match a specified search pattern. The default command gets a list of all open DT trees in the HSI session.

Arguments

`-regexp` – (optional) Specifies that the search <patterns> are written as regular expressions. Both search <patterns> and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `""`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For the "DT tree" object you can use the "DTS_FILE_NAME" property to filter results. The following gets dt trees that do NOT contain the "pl.dtsi" substring within their name:

```
get_dt_trees * -filter {NAME !~ "*pl.dtsi*"}
```

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

<patterns> - (optional) Match DT trees against the specified patterns. The default pattern is the wildcard `*` which gets all DT trees. More than one pattern can be specified to find multiple trees based on different search criteria.

Examples

Get all created DT trees in the current session:

```
hsi::get_dt_trees
```

hsi::get_intf_nets

Description

Get a list of interface nets.

Syntax

```
get_intf_nets [-regexp] [-filter <arg>] [-boundary_type <arg>] [-hierarchical] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Interface Net objects. Returns nothing if the command fails.

Usage

Name	Description
<code>[-regexp]</code>	Patterns are full regular expressions
<code>[-filter]</code>	Filter list with expression
<code>[-boundary_type]</code>	Used when source object is on a hierarchical block's pin. Valid values are 'upper', 'lower', or 'both'. If 'lower' boundary, searches from the lower level of hierarchy onwards. This option is only valid for <code>connected_to</code> relations. Default: upper
<code>[-hierarchical]</code>	Get interface nets from all levels of hierarchical cells
<code>[-of_objects]</code>	Get 'intf_net' objects of these types: 'hw_design cell bus_intf'.
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code>[<patterns>]</code>	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of interface nets in the current hardware design that match a specified search pattern. The default command gets a list of all interface nets in the subsystem design.

Arguments

`-regexp` - (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `. *` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` - (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `" "`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`= =`), `not-equal` (`! =`), `match` (`= ~`), and `not-match` (`! ~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For hardware design nets you can use the "NAME" property to filter results.

`-hierarchical` - (optional) Get interface nets from all levels of hierarchical cells.

`-boundary_type` - (optional) Used when source object is on a hierarchical block's pin. Valid values are 'upper', 'lower', or 'both'. If 'lower' boundary, searches from the lower level of hierarchy onwards. This option is only valid for `connected_to` relations.

`-of_objects <args>` - (optional) Get a list of the nets connected to the specified IP integrator subsystem cell, pin, or port objects.

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_cells` or `get_pins`, rather than specifying objects by name. In addition, `-of_objects` cannot be used with a search pattern.

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` – (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`<patterns>` – (optional) Match hardware design interface nets against the specified patterns. The default pattern is the wildcard `'*'` which returns a list of all interface nets in the current IP integrator subsystem design. More than one pattern can be specified to find multiple nets based on different search criteria.

Note: You must enclose multiple search patterns in braces {} to present the list as a single element.

Examples

The following example gets the interface net attached to the specified pin of an hardware design, and returns the net:

```
hsi::get_intf_nets -of_objects [get_pins aclk]
```

Note: If there are no interface nets matching the pattern you will get a warning.

hsi::get_intf_pins

Description

Get a list of interface pins.

Syntax

```
get_intf_pins [-regexp] [-filter <arg>] [-hierarchical] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Interface pin objects. Returns nothing if the command fails.

Usage

Name	Description
<code>[-regexp]</code>	Patterns are full regular expressions
<code>[-filter]</code>	Filter list with expression
<code>[-hierarchical]</code>	Get interface pins from all levels of hierarchical cells
<code>[-of_objects]</code>	Get 'bus_intf' objects of these types: 'hw_design cell port intf_net'.
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code>[<patterns>]</code>	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of pin objects in the current design that match a specified search pattern. The default command gets a list of all pins in the design.

Arguments

`-regexp` – (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `. *` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `""`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For the interface pins, "NAME" and "TYPE" are some of the properties you can use to filter results. The following gets slave interface pins that do NOT contain the "S_AXI" substring within their name:

```
get_intf_pins * -filter {TYPE == SLAVE && NAME !~ "*S_AXI* "}
```

`-hierarchical` - (optional) Get interface pins from all levels of hierarchical cells.

`-of_objects <arg>` - (optional) Get the pins connected to the specified cell, clock, timing path, or net; or pins associated with specified DRC violation objects.

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_cells` or `get_pins`, rather than specifying objects by name. In addition, `-of_objects` cannot be used with a search `<pattern>`

`-match_style [sdc | ucf]` - (optional) Indicates that the search pattern matches UCF constraints or SDC constraints. The default is SDC.

`-quiet` - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`patterns` - (optional) Match pins against the specified patterns. The default pattern is the wildcard `^*` which gets a list of all pins in the project. More than one pattern can be specified to find multiple pins based on different search criteria.

Note: You must enclose multiple search patterns in braces, {}, or quotes, "", to present the list as a single element.

Examples

The following example gets a list of pins attached to the specified cell:

```
hsi::get_intf_pins -of_objects [lindex [get_cells] 1]
```

Note: If there are no pins matching the pattern, the tool will return a warning.

hsi::get_intf_ports

Description

Get a list of interface ports.

Syntax

```
get_intf_ports [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Interface Port objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get 'bus_intf' objects of these types: 'hw_design port intf_net'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of interface port objects in the current hardware subsystem design that match a specified search pattern. The default command gets a list of all interface ports in the subsystem design.

The external connections in an IP subsystem design are ports, or interface ports. The external connections in an IP integrator cell, or hierarchical module, are pins and interface pins. Use the `get_pins` and `get_intf_pins` commands to select the pin objects.

Note: To improve memory and performance, the `get_*` commands return a container list of a single type of objects (e.g. cells, nets, pins, or ports). You can add new objects to the list (using `lappend` for instance), but you can only add the same type of object that is currently in the list. Adding a different type of object, or string, to the list is not permitted and will result in a Tcl error.

Arguments

`-regexp` – (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `""`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For IP subsystem interface ports, "DIRECTION", and "NAME" are some of the properties you can use to filter results.

`-of_objects <arg>` – (optional) Get the interface ports connected to the specified IP subsystem interface nets returned by `get_intf_nets`.

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_cells` or `get_pins`, rather than specifying objects by name. In addition, `-of_objects` cannot be used with a search `<pattern>`.

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` – (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`patterns` - (optional) Match interface ports against the specified patterns. The default pattern is the wildcard `*` which gets a list of all interface ports in the subsystem design. More than one pattern can be specified to find multiple interface ports based on different search criteria.

Note: You must enclose multiple search patterns in braces {} to present the list as a single element.

Examples

The following example gets the interface ports in the subsystem design that operate in Master mode:

```
hsi::get_intf_ports -filter {MODE=="master"}
```

Note: If there are no interface ports matching the pattern, the tool will return a warning.

hsi::get_mem_ranges

Description

Get a list of memory ranges.

Syntax

```
get_mem_ranges [-regexp] [-filter <arg>] [-hierarchical] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Memory range objects. Returns nothing if the command fails.

Usage

Name	Description
<code>[-regexp]</code>	Patterns are full regular expressions
<code>[-filter]</code>	Filter list with expression
<code>[-hierarchical]</code>	Get memory ranges from all levels of hierarchical cells
<code>[-of_objects]</code>	Get 'mem_range' objects of these types: 'cell'.
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code>[<patterns>]</code>	Match cell names against patterns Default: *

Categories

Hardware

Description

Get a list of slaves of the processor in the current hardware design.

Arguments

`-regexp` - (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` - (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `""`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

`-hierarchical` - (optional) Get memory ranges from all levels of hierarchical cells.

`-of_objects <arg>` - (optional) Get the slaves of the specified object.

`-quiet` - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

patterns - (optional) Match address segments against the specified patterns. The default pattern is the wildcard `*` which gets a list of all address segments in the current IP subsystem design. More than one pattern can be specified to find multiple address segments based on different search criteria.

Note: You must enclose multiple search patterns in braces {} to present the list as a single element.

Examples

The following example gets the slaves of the processor:

```
hsi::get_mem_ranges

hsi::get_mem_ranges -of_objects [lindex [get_cells -filter {IP_TYPE==PROCESSOR}] 0]
```

Note: If there are no objects matching the pattern you will get a warning.

hsi::get_nets

Description

Get a list of nets.

Syntax

```
get_nets [-regexp] [-filter <arg>] [-boundary_type <arg>] [-hierarchical] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Net objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-boundary_type]	Used when source object is on a hierarchical block's pin. Valid values are 'upper', 'lower', or 'both'. If 'lower' boundary, searches from the lower level of hierarchy onwards. This option is only valid for connected_to relations. Default: upper
[-hierarchical]	Get nets from all levels of hierarchical cells
[-of_objects]	Get 'net' objects of these types: 'hw_design cell port'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of nets in the current hardware design that match a specified search pattern. The default command gets a list of all nets in the subsystem design.

Arguments

`-regexp` – (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `""`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For the "hardware design" object you can use the "NAME" property to filter results.

`-boundary_type` – (optional) Used when source object is on a hierarchical block's pin. Valid values are 'upper', 'lower', or 'both'. If 'lower' boundary, searches from the lower level of hierarchy onwards. This option is only valid for `connected_to` relations. Default: upper.

`-hierarchical` – (optional) Get nets from all levels of hierarchical cells.

`-of_objects` – (optional) Get 'net' objects of these types: 'hw_design cell port'.

-quiet – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose – (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

<patterns – (optional) Match hardware design nets against the specified patterns. The default pattern is the wildcard `**` which returns a list of all nets in the current IP integrator subsystem design. More than one pattern can be specified to find multiple nets based on different search criteria.

Note: You must enclose multiple search patterns in braces {} to present the list as a single element.

Examples

The following example gets the net attached to the specified pin of an hardware design module, and returns both the net:

```
hsi::get_nets -of_objects [get_pins aclk]
```

Note: If there are no nets matching the pattern you will get a warning.

hsi::get_nodes

Description

Get a list of child nodes.

Syntax

```
get_nodes [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Node objects. Returns nothing if the command fails.

Usage

Name	Description
<code>[-regexp]</code>	Patterns are full regular expressions
<code>[-filter]</code>	Filter list with expression

Name	Description
<code>[-of_objects]</code>	Get 'node' objects of these types: 'driver sw_proc os node'.
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code>[<patterns>]</code>	Match cell names against patterns Default: *

Categories

Software

Description

Get a list of nodes in drivers/os/nodes in the current software design.

A node can have child nodes in it.

Arguments

`-regexp` – (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `""`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`= =`), `not-equal` (`! =`), `match` (`= ~`), and `not-match` (`! ~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

The following gets nodes that matches NAME and PARENT within their name:

```
get_nodes -filter {NAME==clkc && PARENT == ps7_s1cr_0}

-of_objects <arg> - (optional) Get 'node' objects of these types: 'sw_driver', 'sw_os',
'sw_proc', 'sw_node'.
```

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_nodes`, rather than specifying objects by name. In addition, `-of_objects` cannot be used with a search `<pattern>`.

`-quiet` - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`patterns` - (optional) Match software design cells against the specified patterns. The default pattern is the wildcard `*` which gets a list of all cells in the current IP subsystem design. More than one pattern can be specified to find multiple cells based on different search criteria.

Note: You must enclose multiple search patterns in braces, {}, to present the list as a single element.

Examples

The following example gets a list of nodes that include the specified driver in the software design:

```
hsi::get_nodes -of_objects [get_drivers ps7_uart_0]
```

The following example gets a list of all nodes of OS:

```
hsi::get_nodes -of_objects [get_os]
```

hsi::get_pins

Description

Get a list of pins.

Syntax

```
get_pins [-regexp] [-filter <arg>] [-hierarchical] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Pin objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-hierarchical]	Get pins from all levels of hierarchical cells
[-of_objects]	Get 'port' objects of these types: 'hw_design cell bus_intf net'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of pin objects on the current hardware design that match a specified search pattern. The default command gets a list of all pins in the subsystem design.

Arguments

-regexp – (optional) Specifies that the search <patterns> are written as regular expressions. Both search <patterns> and **-filter** expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

-filter <args> – (optional) Filter the results list with the specified expression. The **-filter** argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard * character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the * wildcard character, this matches a property with a defined value of "".

For pins, "DIR" and "TYPE" are some of the properties you can use to filter results. The following gets input pins that do NOT contain the "RESET" substring within their name:

```
get_pins * -filter {DIRECTION == IN && NAME !~ "*RESET* "}
```

-hierarchical - (optional) Get pins from all levels of hierarchical cells.

-of_objects <arg> - (optional) Get the pins connected to the specified IP subsystem cell or net.

Note: The -of_objects option requires objects to be specified using the `get_*` commands, such as `get_cells` or `get_pins`, rather than specifying objects by name. In addition, -of_objects cannot be used with a search <pattern>.

-quiet - (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

-verbose - (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

patterns - (optional) Match hardware design pins against the specified patterns.

Note: More than one pattern can be specified to find multiple pins based on different search criteria. You must enclose multiple search patterns in braces {} to present the list as a single element.

Examples

The following example gets a list of pins attached to the specified cell:

```
hsi::get_pins -of [get_cells axi_gpio_0]
```

Note: If there are no pins matching the pattern, the tool will return a warning.

The following example gets a list of pins attached to the specified subsystem net:

```
hsi::get_pins -of [get_nets ps7_axi_interconnect_0_M_AXI_BRESP]
```

hsi::get_ports

Description

Get a list of external ports.

Syntax

```
get_ports [-regexp] [-filter <arg>] [-of_objects <args>] [-quiet] [-verbose] [<patterns>...]
```

Returns

Port objects. Returns nothing if the command fails.

Usage

Name	Description
[-regexp]	Patterns are full regular expressions
[-filter]	Filter list with expression
[-of_objects]	Get 'port' objects of these types: 'hw_design bus_intf net'.
[-quiet]	Ignore command errors
[-verbose]	Suspend message limits during command execution
[<patterns>]	Match cell names against patterns Default: *

Categories

Hardware

Description

Gets a list of port objects in the current hardware design that match a specified search pattern. The default command gets a list of all ports in the hardware design.

The external connections in an hardware design are ports, or interface ports. The external connections in an IP integrator cell, or hierarchical module, are pins and interface pins. Use the `get_pins` and `get_intf_pins` commands to select the pin objects.

Arguments

`-regexp` – (optional) Specifies that the search `<patterns>` are written as regular expressions. Both search `<patterns>` and `-filter` expressions must be written as regular expressions when this argument is used. Xilinx regular expression Tcl commands are always anchored to the start of the search string. You can add `.*` to the beginning or end of a search string to widen the search to include a substring. See [this web page](#) for help with regular expression syntax.

Note: The Tcl built-in command `regexp` is not anchored, and works as a standard Tcl command. For more information, refer to [this web page](#).

`-filter <args>` – (optional) Filter the results list with the specified expression. The `-filter` argument filters the list of objects returned based on property values on the objects. You can find the properties on an object with the `report_property` or `list_property` commands.

Quote the filter search pattern to avoid having to escape special characters that might be found in net, pin, or cell names, or other properties. String matching is case sensitive and is always anchored to the start and to the end of the search string. The wildcard `*` character can be used at the beginning or at the end of a search string to widen the search to include a substring of the property value.

Note: The filter returns an object if a specified property exists on the object, and the specified pattern matches the property value on the object. In the case of the `*` wildcard character, this matches a property with a defined value of `" "`.

For string comparison, the specific operators that can be used in filter expressions are `equal` (`==`), `not-equal` (`!=`), `match` (`=~`), and `not-match` (`!~`). Numeric comparison operators `<`, `>`, `<=`, and `>=` can also be used. Multiple filter expressions can be joined by `AND` and `OR` (`&&` and `||`).

For IP subsystem ports, "DIRECTION", "TYPE", and "SENSITIVITY" are some of the properties you can use to filter results.

`-of_objects <arg>` – (optional) Get the ports connected to the specified IP subsystem nets returned by `get_nets`.

Note: The `-of_objects` option requires objects to be specified using the `get_*` commands, such as `get_cells` or `get_pins`, rather than specifying objects by name. In addition, `-of_objects` cannot be used with a search `<pattern>`.

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` – (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`patterns` - (optional) Match ports against the specified patterns. The default pattern is the wildcard `**` which gets a list of all ports in the subsystem design. More than one pattern can be specified to find multiple ports based on different search criteria.

Note: You must enclose multiple search patterns in braces {} to present the list as a single element.

Examples

The following example gets the ports connected to the specified hardware subsystem net:

```
hsi::get_ports -of_objects [get_nets bridge_1_apb_m] -filter {DIRECTION==I}
```

Note: If there are no ports matching the pattern, the tool will return a warning.

hsi::open_hw_design

Description

Open a hardware design from disk file.

Syntax

```
open_hw_design [-quiet] [-verbose] [<file>]
```

Returns

Hardware design object. Returns nothing if the command fails.

Usage

Name	Description
<code>[-quiet]</code>	Ignore command errors
<code>[-verbose]</code>	Suspend message limits during command execution
<code><file></code>	Hardware design file to open

Categories

Hardware

Description

Opens a Hardware design in the Hardware Software Interface. The hardware design must be exported previously using the Vivado product. Users can open multiple hardware designs at same time.

If successful, this command returns a hardware design object representing the opened Hardware design. Otherwise it returns an error.

Arguments

`-quiet` – (optional) Execute the command quietly, returning no messages from the command. The command also returns `TCL_OK` regardless of any errors encountered during execution.

Note: Any errors encountered on the command line while launching the command are returned. Only errors occurring inside the command are trapped.

`-verbose` – (optional) Temporarily override any message limits and return all messages from this command.

Note: Message limits can be defined with the `set_msg_config` command.

`file` – The path and file name of the Hardware design to open in the HSM. The name must include the file extension.

Examples

Open the specified IP subsystem design in the current project:

```
open_hw_design C:/Data/project1/project1.sdk/SDK/SDK_Export/hw/design_1.xml
```

OR

```
open_hw_design C:/Data/project1/project1.sdk/design_1_wrapper.xsa
```

GNU Compiler Tools

This section contains the following chapters:

- [Overview](#)
- [Compiler Framework](#)
- [Common Compiler Usage and Options](#)
- [MicroBlaze Compiler Usage and Options](#)
- [Arm Compiler Usage and Options](#)
- [Other Notes](#)

Note: From 2024.1 release, this user guide supports MicroBlaze™ and MicroBlaze™-V devices.

Overview

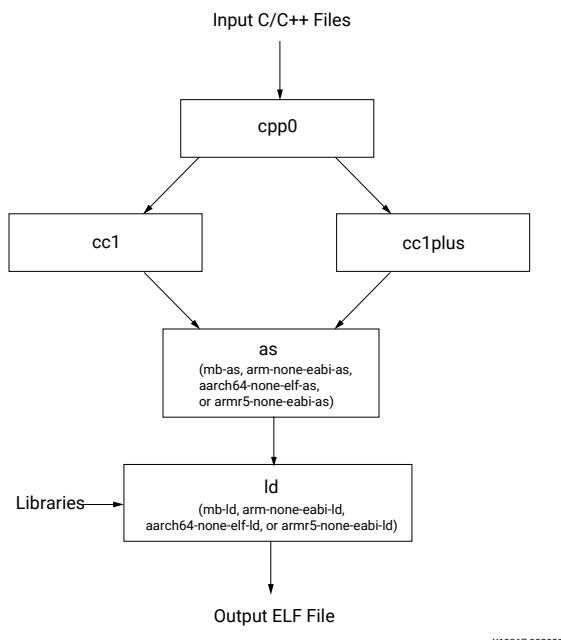
The AMD Vivado™ Design Suite includes the GNU compiler collection (GCC) for the MicroBlaze™ and MicroBlaze™-V processor and the Arm® Cortex® A9, A53, A72, and R5 processors.

- The Vivado GNU tools support both the C and C++ languages.
- The MicroBlaze and MicroBlaze™-V GNU tools include the `mb-gcc` and `mb-g++` compilers, the `mb-as` assembler, and the `mb-ld` linker.
- The Arm processor tools include:
 - The `arm-none-eabi-gcc` and `arm-none-eabi-g++` compilers, `arm-none-eabi-as` assembler, and `arm-none-eabi-ld` linker for Cortex A9 processors
 - `aarch64-none-elf-*` for Cortex-A53 and Cortex-A72 processors
 - `armr5-none-eabi-*` for Cortex-R5F processors
- The toolchains also include the C, math, GCC, and C++ standard libraries.
- The compiler also uses the common binary utilities (referred to as binutils), such as an assembler, a linker, and object dump. The MicroBlaze and Arm compiler tools use the GNU binutils based on GNU version 2.31 in 2019.x and 2.32 in 2020.x of the sources.

Compiler Framework

This section discusses the common features of the MicroBlaze™ and MicroBlaze™-V, and Cortex® A9, A53, A72, and R5 processor compilers. The following figure displays the GNU tool flow.

Figure 74: GNU Tool Flow



X13367-082021

The GNU compilers are named as follows:

- `mb-gcc` for MicroBlaze and MicroBlaze™-V
- `arm-none-eabi-gcc` for Cortex A9 cores
- `aarch64-none-elf-gcc` for Cortex-A53 and Cortex-A72
- `armr5-none-eabi-gcc` for Cortex-R5F

The GNU compiler is a wrapper that calls the following executables:

- **Pre-processor (`cpp0`):** This is the first pass invoked by the compiler. The pre-processor replaces all macros with definitions as defined in the source and header files.

- **Machine and language specific compiler:** This compiler works on the pre-processed code, which is the output of the first stage. The language-specific compiler is one of the following:
 - **C Compiler (cc1):** The compiler responsible for most of the optimizations done on the input C code and for generating assembly code.
 - **C++ Compiler (cc1plus):** The compiler responsible for most of the optimizations done on the input C++ code and for generating assembly code.
- **Assembler:** The assembly code has mnemonics in assembly language. The assembler converts these to machine language. The assembler also resolves some of the labels generated by the compiler. It creates an object file which is passed on to the linker. The assembler executables are named as follows:
 - `mb-as` for MicroBlaze and MicroBlaze™-V
 - `arm-none-eabi-as` for Cortex A9 cores
 - `aarch64-none-elf-as` for Cortex-A53 and Cortex-A72
 - `armr5-none-eabi-as` for Cortex-R5F
- **Linker:** Links all the object files generated by the assembler. If libraries are provided on the command line, the linker resolves some of the undefined references in the code by linking in some of the functions from the assembler. The linker executables named as follows:
 - `mb-ld` for MicroBlaze
 - `arm-none-eabi-ld` for Cortex A9 cores
 - `aarch64-none-elf-ld` for Cortex-A53 and Cortex-A72
 - `armr5-none-eabi-ld` for Cortex-R5F

Executable options are described in the following sections:

- [Commonly Used Compiler Options: Quick Reference](#)
- [Linker Options](#)
- [MicroBlaze Compiler Usage and Options](#)
- [MicroBlaze Linker Options](#)
- [Arm Compiler Usage and Options](#)

Note: From this point forward, the references to GCC in this chapter refer to the MicroBlaze and MicroBlaze™-V compiler, `mb-gcc`, and references to G++ refer to the MicroBlaze C++ compiler, `mb-g++`.

Common Compiler Usage and Options

Usage

To use the GNU compiler, type:

```
<Compiler_Name> options files...
```

Where **<Compiler_Name>** refers to one of the following compilers:

- `mb-gcc` for MicroBlaze and MicroBlaze™-V
- `arm-none-eabi-gcc` for Cortex A9
- `aarch64-none-elf-gcc` for Cortex-A53 and Cortex-A72
- `armr5-none-eabi-gcc` for Cortex-R5F

To compile C++ programs, you can use the `mb-g++` or `arm-none-eabi-g++` command.

Input Files

The compilers take one or more of the following files as input:

- C source files
- C++ source files
- Assembly files
- Object files
- Linker scripts

Note: These files are optional. If they are not specified, the default linker script embedded in the linker is used. The default scripts are as follows:

- `mb-ld` for MicroBlaze and MicroBlaze™-V

- `arm-none-eabi-ld` for Cortex A9
- `aarch64-none-elf-ld` for Cortex-A53 and Cortex-A72
- `armr5-none-eabi-ld` for Cortex-R5F

The default extensions for each of these types are listed in [File Types and Extensions](#). In addition to the files mentioned above, the compiler implicitly refers to the libraries files `libc.a`, `libgcc.a`, `libm.a`, and `libxil.a`. The default location for these files is the Vivado installation directory. When using the G++ Compiler, the `libsupc++.a` and `libstdc++.a` files are also referenced. These are the C++ language support and C++ platform libraries respectively.

Output Files

The compiler generates the following files as output:

- An ELF file.
 - An assembly file, if the `-save-temp`s or `-S` option is used.
 - An object file, if the `-save-temp`s or `-c` option is used.
 - Preprocessor output, an `.i` or `.ii` file, if the `-save-temp`s option is used.
-

File Types and Extensions

The GNU compiler determines the type of your file from the file extension. The following table lists the valid extensions and the corresponding file types. The GCC wrapper calls the appropriate lower level tools by recognizing these file types.

Table 60: File Extensions

Extension	File Type (Dialect)
<code>.c</code>	C file
<code>.C</code>	C++ file
<code>.cxx</code>	C++ file
<code>.cpp</code>	C++ file
<code>.c++</code>	C++ file
<code>.cc</code>	C++ file
<code>.S</code>	Assembly file, but might have preprocessor directives
<code>.s</code>	Assembly file with no preprocessor directives

Libraries

The following table lists the libraries necessary for the `mb_gcc` and `arm-none-eabi-gcc` compilers.

Table 61: Libraries Used by the Compilers

Library	Particular
<code>libxil.a</code>	Contains drivers, software services (such as XiIMFS), and initialization files developed for the Vivado tools.
<code>libc.a</code>	Standard C libraries, including functions such as <code>strcmp</code> and <code>strlen</code> .
<code>libgcc.a</code>	GCC low-level library containing emulation routines for floating point and 64-bit arithmetic.
<code>libm.a</code>	Math library containing functions such as <code>cos</code> and <code>sine</code> .
<code>libsupc++.a</code>	C++ support library with routines for exception handling, RTTI, and others.
<code>libstdc++.a</code>	C++ standard platform library. Contains standard language classes, such as those for stream I/O, file I/O, string manipulation, and others.

Libraries are linked in automatically by both compilers. If the standard libraries are overridden, the search path for these libraries must be given to the compiler. The `libxil.a` is modified to add driver and library routines.

Language Dialect

The GCC compiler recognizes both C and C++ dialects and generates code accordingly. By GCC convention, it is possible to use either the GCC or the G++ compilers equivalently on a source file. The compiler that you use and the extension of your source file determines the dialect used on the input and output files.

When using the GCC compiler, the dialect of a program is always determined by the file extension, as listed in [File Types and Extensions](#). If a file extension shows that it is a C++ source file, the language is set to C++. This means that if you have compile C code contained in a CC file, even if you use the GCC compiler, it automatically mangles function names.

The primary difference between GCC and G++ is that G++ automatically sets the default language dialect to C++ (irrespective of the file extension), and if linking, automatically pulls in the C++ support libraries. This means that even if you compile C code in a `.c` file with the G++ compiler, it will mangle names.

Name mangling is a concept unique to C++ and other languages that support overloading of symbols. A function is said to be overloaded if the same function can perform different actions based on the arguments passed in, and can return different return values. To support this, C++ compilers encode the type of the function to be invoked in the function name, avoiding multiple definitions of a function with the same name.

Be careful about name mangling if you decide to follow a mixed compilation mode, with some source files containing C code and some others containing C++ code (or using GCC for compiling certain files and G++ for compiling others). To prevent name mangling of a C symbol, you can use the following construct in the symbol declaration.

```
#ifdef __cplusplus
extern "C" {
#endif
int foo();
int morefoo();
#endif
#endif
```

Make these declarations available in a header file and use them in all source files. This causes the compiler to use the C dialect when compiling definitions or references to these symbols.

Note: All Vivado drivers and libraries follow these conventions in all the header files they provide. You must include the necessary headers, as documented in each driver and library, when you compile with G++. This ensures that the compiler recognizes library symbols as belonging to "C" type.

When compiling with either variant of the compiler, to force a file to a particular dialect, use the `-x lang` switch. Refer to the [GCC documentation](#) for more information on this switch.

- When using the GCC compiler, `libstdc++.a` and `libsupc++.a` are not automatically linked in.
- When compiling C++ programs, use the G++ variant of the compiler to make sure all the required support libraries are linked in automatically.
- Adding `-lstdc++` and `-lsupc++` to the GCC command are also possible options.

For more information about how to invoke the compiler for different languages, refer to the [GNU online documentation](#).

Commonly Used Compiler Options: Quick Reference

The summary below lists compiler options that are common to the compilers for MicroBlaze and MicroBlaze™-V, and Arm processors. The compiler options are case sensitive.

General Options

- **-E:** Preprocess only; do not compile, assemble and link. The preprocessed output displays on the standard out device.
- **-S:** Compile only; do not assemble and link. Generates a `.s` file.
- **-c:** Compile and assemble only; do not link. Generates a `.o` file.
- **-g:** This option adds DWARF2-based debugging information to the output file. The debugging information is required by the GNU debugger, mb-gdb or arm-none-eabi-gdb. The debugger provides debugging at the source and the assembly level. This option adds debugging information only when the input is a C/C++ source file.
- **-gstabs:** Use this option for adding STABS-based debugging information on assembly (`.S`) files and assembly file symbols at the source level. This is an assembler option that is provided directly to the GNU assembler, mb-as or arm-none-eabi-as. If an assembly file is compiled using the compiler mb-gcc or arm-none-eabi-gcc, prefix the option with `-Wa`.
- **-On:** The GNU compiler provides optimizations at different levels. The optimization levels in the following table apply only to the C and C++ source files.

Table 62: Optimizations for Values of n

n	Optimization
0	No optimization.
1	Medium optimization.
2	Full optimization
3	Full optimization. Attempt automatic inlining of small subprograms.
5	Optimize for size.

Note: Optimization levels 1 and above cause code re-arrangement. While debugging your code, use of no optimization level is recommended. When an optimized program is debugged through GDB, the displayed results might seem inconsistent.

- **-v:** This option executes the compiler and all the tools underneath the compiler in verbose mode. This option gives complete description of the options passed to all the tools. This description is helpful in discovering the default options for each tool.
- **-save-temp:** The GNU compiler provides a mechanism to save the intermediate files generated during the compilation process. The compiler stores the following files:
 - Preprocessor output `-input_file_name.i` for C code and `input_file_name.ii` for C++ code
 - Compiler (cc1) output in assembly format `-input_file_name.s`
 - Assembler output in ELF format `-input_file_name.s`

The compiler saves the default output of the entire compilation as a.out.

- **-o filename**: The compiler stores the default output of the compilation process in an ELF file named a.out. You can change the default name using -o output_file_name. The output file is created in ELF format.
- **-Wp,<option>, -Wa,<option>, and -Wl,<option>**: The compiler, `mb-gcc` or `arm-none-eabi-gcc`, is a wrapper around other executables such as the preprocessor, compiler (`cc1`), assembler, and the linker. You can run these components of the compiler individually or through the top level compiler.

There are certain options that are required by tools, but might not be necessary for the top-level compiler. To run these commands, use the options listed in the following table.

Table 63: Tool-Specific Options Passed to the Top-Level GCC Compiler

Option	Tool	Example
<code>-Wp,<option></code>	Preprocessor	<code>mb-gcc -Wp, -D -Wp, MYDEFINE ...</code> Signal the pre-processor to define the symbol MYDEFINE with the -D MYDEFINE option.
<code>-Wa,<option></code>	Assembler	<code>mb-as -Wa, ...</code> Signal the assembler to target the MicroBlaze and MicroBlaze™-V processor.
<code>-Wl,<option></code>	Linker	<code>mb-gcc -Wl, -M ...</code> Signal the linker to produce a map file with the -M option.

- **-help**: Use this option with any GNU compiler to get more information about the available options. You can also consult the GCC manual.
- **-B directory**: Add directory to the C runtime library search paths.
- **-L directory**: Add directory to the library search path.
- **-I directory**: Add directory to header search path.
- **-l library**: Search library for undefined symbols.

Note: The compiler prefixes “lib” to the library name indicated in this command line switch.

Library Search Options

- `-l <library name>`: By default, the compiler searches only the standard libraries, such as `libc`, `libm`, and `libxil`. You can also create your own libraries. You can specify the name of the library and where the compiler can find the definition of these functions. The compiler prefixes `lib` to the library name that you provide.

The compiler is sensitive to the order in which you provide options, particularly the `-l` command line switch. Provide this switch only after all of the sources in the command line.

For example, if you create your own library called `libproject.a`. you can include functions from this library using the following command:

```
Compiler Source_Files -L${LIBDIR} -l project
```

 **IMPORTANT!** If you supply the library flag `-l library_name` before the source files, the compiler does not find the functions called from any of the sources. This is because the compiler search is only done in one direction and it does not keep a list of available libraries.

- `-L <lib directory>`: This option indicates the directories in which to search for the libraries. The compiler has a default library search path, where it looks for the standard library. Using the `-L` option, you can include some additional directories in the compiler search path.

Header File Search Option

- `-I <directory name>`: This option searches for header files in the `/<dir_name>` directory before searching the header files in the standard path.

Default Search Paths

The compilers (mb-gcc for MicroBlaze and MicroBlaze™-V, arm-none-eabi-gcc for Cortex A9, armr5-none-eabi-gcc for Cortex-R5F, and aarch64-none-elf-gcc for Cortex-A53 Cortex®-A72) search certain paths for libraries and header files. The search paths on the various platforms are described below.

Library Search Procedures

The compilers search libraries in the following order:

1. Directories are passed to the compiler with the `-L <dir_name>` option.

2. The compilers search the following libraries:

```
$XILINX_VITIS/gnu/<processor>/<platform>/<processor-lib>/usr/lib
```

Header File Search Procedures

The compilers search header files in the following order:

1. Directories are passed to the compiler with the `-I <dir_name>` option.
2. The compilers search the following header files:

```
$XILINX_VITIS/gnu/<processor>/<platform>/<processor-lib>/usr/include
```

Initialization File Search Procedures

The compilers search initialization files in the following order:

1. Directories are passed to the compiler with the `-B <dir_name>` option.
2. The compilers search the following files:

```
$XILINX_VITIS/gnu/<processor>/<platform>/<processor-lib>/usr/lib
```

Where:

- `<processor>` is `microblaze` for MicroBlaze and MicroBlaze™-V processors, `aarch32` for A9, `aarch64` for A53, and `armr5` for R5 processors.
- `<processor-lib>` is `microblazeeb-xilinx-elf` for MicroBlaze and MicroBlaze™-V, `aarch32-xilinx-eabi` for A9, `aarch64-none/aarch64-xilinx-elf` for A53, and `gcc-arm-none-eabi/armrm-xilinx-eabi` for R5 processors.

Note: `platform` indicates `lin64` for Linux 64-bit and `nt` for Windows Cygwin.

Linker Options

- `-defsym _STACK_SIZE=value`: The total memory allocated for the stack can be modified using this linker option. The variable `_STACK_SIZE` is the total space allocated for the stack. The `_STACK_SIZE` variable is given the default value of 100 words, or 400 bytes. If your program is expected to need more than 400 bytes for stack and heap combined, it is recommended that you increase the value of `_STACK_SIZE` using this option. The value is in bytes.

In certain cases, a program might need a bigger stack. If the stack size required by the program is greater than the stack size available, the program tries to write in other, incorrect, sections of the program, leading to incorrect execution of the code.

Note: A minimum stack size of 16 bytes (0x0010) is required for programs linked with the AMD-provided C runtime (CRT) files.

- **-f `defsym _HEAP_SIZE=value`**: The total memory allocated for the heap can be controlled by the value given to the variable `_HEAP_SIZE`. The default value of `_HEAP_SIZE` is zero.

Dynamic memory allocation routines use the heap. If your program uses the heap in this fashion, then you must provide a reasonable value for `_HEAP_SIZE`.

For advanced users: you can generate linker scripts directly from IP integrator.

Memory Layout

The MicroBlaze and MicroBlaze™-V and Arm Cortex A9 and R5 processors use 32-bit logical addresses and can address any memory in the system in the range 0x0 to 0xFFFFFFFF. This address range can be categorized into reserved memory and I/O memory. The Arm Cortex-A53 and Cortex-A72 processor uses 64-bit logical addresses.

Reserved Memory

Reserved memory has been defined by the hardware and software programming environment for privileged use. This is typically true for memory containing interrupt vector locations and operating system level routines. The following table lists the reserved memory locations for MicroBlaze, MicroBlaze™-V, and Arm processors as defined by the processor hardware. For more information on these memory locations, refer to the corresponding processor reference manuals.

For information about the Arm Cortex A9 memory map, refer to the . For the Cortex-R5F, Cortex-A53, and Cortex-A72 memory map, refer to the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Note: In addition to these memories that are reserved for hardware use, your software environment can reserve other memories. Refer to the manual of the particular software platform that you are using to find out if any memory locations are deemed reserved.

Table 64: Hardware Reserved Memory Locations

Processor Family	Reserved Memories	Reserved Purpose	Default Text Start Address
MicroBlaze and MicroBlaze™-V	0x0 - 0x4F	Reset, Interrupt, Exception, and other reserved vector locations.	0x50

I/O Memory

I/O memory refers to addresses used by your program to communicate with memory-mapped peripherals on the processor buses. These addresses are defined as a part of your hardware platform specification.

User and Program Memory

User and program memory refers to all the memory that is required for your compiled executable to run. By convention, this includes memories for storing instructions, read-only data, read-write data, program stack, and program heap. These sections can be stored in any addressable memory in your system. By default the compiler generates code and data starting from the address listed in the table in [Reserved Memory](#) and occupying contiguous memory locations. This is the most common memory layout for programs. You can modify the starting location of your program by defining (in the linker) the symbol `_TEXT_START_ADDR` for MicroBlaze and MicroBlaze™-V, `START_ADDR` for Arm.

In special cases, you might want to partition the various sections of your ELF file across different memories. This is done using the linker command language (refer to [Linker Scripts](#) for details).

The following are some situations in which you might want to change the memory map of your executable:

- When partitioning large code segments across multiple smaller memories
- Remapping frequently executed sections to fast memories
- Mapping read-only segments to non-volatile flash memories

No restrictions apply to how you can partition your executable. The partitioning can be done at the output section level, or even at the individual function and data level. The resulting ELF can be non-contiguous, that is, there can be “holes” in the memory map. Ensure that you do not use documented reserved locations.

Alternatively, if you are an advanced user and want to modify the default binary data provided by the tools for the reserved memory locations, you can do so. In this case, you must replace the default startup files and the memory mappings provided by the linker.

Object-File Sections

An executable file is created by concatenating input sections from the object files (.o files) being linked together. The compiler, by default, creates code across standard and well-defined sections. Each section is named based on its associated meaning and purpose. The various standard sections of the object file are displayed in the following table.

In addition to these sections, you can also create your own custom sections and assign them to memories of your choice.

Table 65: Sectional Layout of an Object or Executable File

Section	Description
.text	Text section
.rodata	Read only data section
.sdata2	Small read only data section
.sbss2	Small read only uninitialized data section
.data	Read/write data section
.sdata	Small read/write data section
.sbss	Small uninitialized data section
.bss	Uninitialized data section
.heap	Program heap memory section
.stack	Program stack memory section

The reserved sections that you would not typically modify include: `.init`, `.fini`, `.ctors`, `.dtors`, `.got`, `.got2`, and `.eh_frame`.

- **.text:** This section of the object file contains executable program instructions. This section has the `x` (executable), `r` (read-only) and `i` (initialized) flags. This means that this section can be assigned to an initialized read-only memory (ROM) that is addressable from the processor instruction bus.
- **.rodata:** This section contains read-only data. This section has the `r` (read-only) and the `i` (initialized) flags. Like the `.text` section, this section can also be assigned to an initialized, read-only memory that is addressable from the processor data bus.
- **.sdata2:** This section is similar to the `.rodata` section. It contains small read-only data of size less than 8 bytes. All data in this section is accessed with reference to the read-only small data anchor. This ensures that all the contents of this section are accessed using a single instruction. You can change the size of the data going into this section with the `-G` option to the compiler. This section has the `r` (read-only) and the `i` (initialized) flags.
- **.data:** This section contains read-write data and has the `w` (read-write) and the `i` (initialized) flags. It must be mapped to initialized random access memory (RAM). It cannot be mapped to a ROM.
- **.sdata:** This section contains small read-write data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. All data in this section is accessed with reference to the read-write small data anchor. This ensures that all contents of the section can be accessed using a single instruction. This section has the `w` (read-write) and the `i` (initialized) flags and must be mapped to initialized RAM.

- **.sbss2:** This section contains small, read-only uninitialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the `r` (read) flag and can be mapped to ROM.
- **.sbss:** This section contains small uninitialized data of a size less than 8 bytes. You can change the size of the data going into this section with the `-G` option. This section has the `w` (read-write) flag and must be mapped to RAM.
- **.bss:** This section contains uninitialized data. This section has the `w` (read-write) flag and must be mapped to RAM.
- **.heap:** This section contains uninitialized data that is used as the global program heap. Dynamic memory allocation routines allocate memory from this section. This section must be mapped to RAM.
- **.stack:** This section contains uninitialized data that is used as the program stack. This section must be mapped to RAM. This section is typically laid out right after the `.heap` section. In some versions of the linker, the `.stack` and `.heap` sections might appear merged together into a section named `.bss_stack`.
- **.init:** This section contains language initialization code and has the same flags as `.text`. It must be mapped to initialized ROM.
- **.fini:** This section contains language cleanup code and has the same flags as `.text`. It must be mapped to initialized ROM.
- **.ctors:** This section contains a list of functions that must be invoked at program startup and the same flags as `.data` and must be mapped to initialized RAM.
- **.dtors:** This section contains a list of functions that must be invoked at program end, the same flags as `.data`, and it must be mapped to initialized RAM.
- **.got2 / .got:** This section contains pointers to program data, the same flags as `.data`, and it must be mapped to initialized RAM.
- **.eh_frame:** This section contains frame unwind information for exception handling. It contains the same flags as `.rodata`, and can be mapped to initialized ROM.
- **.tbss:** This section holds uninitialized thread-local data that contribute to the program memory image. This section has the same flags as `.bss`, and it must be mapped to RAM.
- **.tdata:** This section holds initialized thread-local data that contribute to the program memory image. This section must be mapped to initialized RAM.
- **.gcc_except_table:** This section holds language specific data. This section must be mapped to initialized RAM.
- **.jcr:** This section contains information necessary for registering compiled Java classes. The contents are compiler-specific and used by compiler initialization functions. This section must be mapped to initialized RAM.

- **.fixup:** This section contains information necessary for doing fixup, such as the fixup page table and the fixup record table. This section must be mapped to initialized RAM.

Linker Scripts

The linker utility uses commands specified in linker scripts to divide your program on different blocks of memories. It describes the mapping between all of the sections in all of the input object files to output sections in the executable file. The output sections are mapped to memories in the system. You do not need a linker script if you do not want to change the default contiguous assignment of program contents to memory. There is a default linker script provided with the linker that places section contents contiguously.

You can selectively modify only the starting address of your program by defining the linker symbol `_TEXT_START_ADDR` on MicroBlaze and MicroBlaze™-V processors, or `START_ADDR` on Arm processors, as displayed in this example:

```
mb-gcc <input files and flags> -Wl,-defsym _TEXT_START_ADDR=0x100
mb-ld <.o files> -defsym _TEXT_START_ADDR=0x100
```

The choices of the default script that are used by the linker from the `$XILINX_gnu/<procname>/<platform>/<processor_name>/lib/ldscripts` area are described as follows (where `<procname> = microblaze`, `<processor_name> = microblaze`, and `<platform> = lin or nt`):

- `elf32<procname>.x` is used by default when none of the following cases apply.
- `elf32<procname>.xn` is used when the linker is invoked with the `-n` option.
- `elf32<procname>.xbn` is used when the linker is invoked with the `-N` option.
- `elf32<procname>.xr` is used when the linker is invoked with the `-r` option.
- `elf32<procname>.xu` is used when the linker is invoked with the `-Ur` option.

To use a linker script, provide it on the GCC command line. Use the command line option `-T <script>` for the compiler, as described below:

```
compiler -T <linker_script> <Other Options and Input Files>
```

If the linker is executed on its own, include the linker script as follows:

```
linker -T <linker_script> <Other Options and Input Files>
```

This tells GCC to use your linker script in the place of the default built-in linker script. Linker scripts can be generated for your program by right clicking the application component and selecting **Reset Link Script** to generate or reset the linker script..

Linker scripts can be used to assign specific variables or functions to specific memories. This is done through section attributes in the C code. Linker scripts can also be used to assign specific object files to sections in memory. These and other features of GNU linker scripts are explained in the GNU linker documentation, which is a part of the [binutils manual](#). For a specific list of input sections that are assigned by MicroBlaze and MicroBlaze™-V processor linker scripts, see [MicroBlaze Linker Script Sections](#).

MicroBlaze Compiler Usage and Options

The MicroBlaze™ GNU compiler is derived from the standard GNU sources as the AMD port of the compiler. The features and options that are unique to the MicroBlaze compiler are described in the sections that follow. When compiling with the MicroBlaze compiler, the pre-processor provides the definition `_MICROBLAZE_` automatically. You can use this definition in any conditional code.

MicroBlaze Compiler

The `mb-gcc` compiler for the MicroBlaze soft processor introduces new options as well as modifications to certain options supported by the GNU compiler tools. The new and modified options are summarized in this chapter.

Processor Feature Selection Options

- `-mcpu=vX.YY.Z`: This option directs the compiler to generate code suited to MicroBlaze hardware version v.X.YY.Z. To get the most optimized and correct code for a given processor, use this switch with the hardware version of the processor.

The `-mcpu` switch behaves differently for different versions, as described below:

- **Pr-v3.00.a**: Uses 3-stage processor pipeline mode. Does not inhibit exception causing instructions being moved into delay slots.
- **v3.00.a and v4.00.a**: Uses 3-stage processor pipeline model. Inhibits exception causing instructions from being moved into delay slots.
- **v5.00.a and later**: Uses 5-stage processor pipeline model. Does not inhibit exception causing instructions from being moved into delay slots.

- **-mlittle-endian/-mbig-endian:** Use these options to select the endianness of the target machine for which code is being compiled. The endianness of the binary object file produced is also set appropriately based on this switch. The GCC driver passes switches to the sub tools (as, cc1, cc1plus, ld) to set the corresponding endianness in the sub tool.

The default is `-mbig-endian`.

You cannot link together object files of mixed endianness.

- **-mno-xl-soft-mul:** This option permits use of hardware multiply instructions for 32-bit multiplications. The MicroBlaze processor has an option to turn the use of hardware multiplier resources on or off. This option should be used when the hardware multiplier option is enabled on the MicroBlaze processor. Using the hardware multiplier can improve the performance of your application. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.
- **-mxl-multiply-high:** The MicroBlaze processor has an option to enable instructions that can compute the higher 32bits of a 32x32-bit multiplication. This option tells the compiler to use these multiply high instructions. The compiler automatically defines the C pre-processor definition `HAVE_HW_MUL_HIGH` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is available or not.
- **-mno-xl-multiply-high:** Do not use multiply high instructions. This option is the default.
- **-mxl-soft-mul:** This option tells the compiler that there is no hardware multiplier unit on the MicroBlaze processor, so every 32-bit multiply operation is replaced by a call to the software emulation `routine_muls13`. This option is the default.
- **-mno-xl-soft-div:** You can instantiate a hardware divide unit in MicroBlaze. When the divide unit is present, this option tells the compiler that hardware divide instructions can be used in the program being compiled.

This option can improve the performance of your program if it has a significant amount of division operations. The compiler automatically defines the C pre-processor definition `HAVE_HW_DIV` when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.

- **-mxl-soft-div:** This option tells the compiler that there is no hardware divide unit on the target MicroBlaze hardware.

This option is the default. The compiler replaces all 32-bit divisions with a call to the corresponding software emulation routines (`_divs13`, `_udivs13`).

- **-mxl-barrel-shift:** The MicroBlaze processor can be configured to be built with a barrel shifter. In order to use the barrel shift feature of the processor, use the option `-mxl-barrel-shift`.

The default option assumes that no barrel shifter is present, and the compiler uses add and multiply operations to shift the operands. Enabling barrel shifts can speed up your application significantly, especially while using a floating point library. The compiler automatically defines the C pre-processor definition HAVE_HW_BSHIFT when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether or not this feature is specified as available.

- **-mno-xl-barrel-shift:** This option tells the compiler not to use hardware barrel shift instructions. This option is the default.
- **-mxl-pattern-compare:** This option activates the use of pattern compare instructions in the compiler.

Using pattern compare instructions can speed up boolean operations in your program. Pattern compare operations also permit operating on word-length data as opposed to byte-length data on string manipulation routines such as `strcpy`, `strlen`, and `strcmp`. On a program heavily dependent on string manipulation routines, the speed increase obtained will be significant. The compiler automatically defines the C pre-processor definition HAVE_HW_PCMP when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.

- **-mno-xl-pattern-compare:** This option tells the compiler not to use pattern compare instructions. This is the default.
- **-mhard-float:** This option turns on the usage of single precision floating point instructions (`fadd`, `frsub`, `fmul`, and `fdiv`) in the compiler.

It also uses `fcmp.p` instructions, where `p` is a predicate condition such as `le`, `ge`, `lt`, `gt`, `eq`, `ne`. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware. The compiler automatically defines the C pre-processor definition HAVE_HW_FPU when this switch is used. This allows you to write C or assembly code tailored to the hardware, based on whether this feature is specified as available or not.

- **-msoft-float:** This option tells the compiler to use software emulation for floating point arithmetic. This option is the default.
- **-mxl-float-convert:** This option turns on the usage of single precision floating point conversion instructions (`fint` and `flt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.
- **-mxl-float-sqrt:** This option turns on the usage of single precision floating point square root instructions (`fsqrt`) in the compiler. These instructions are natively decoded and executed by MicroBlaze, when the FPU is enabled in hardware and these optional instructions are enabled.

General Program Options

- **-msmall-divides:** This option generates code optimized for small divides when no hardware divider exists. For signed integer divisions where the numerator and denominator are between 0 and 15 inclusive, this switch provides very fast table-lookup-based divisions. This switch has no effect when the hardware divider is enabled.
- **-mxl-gp-opt:** If your program contains addresses that have non-zero bits in the most significant half (top 16 bits), then load or store operations to that address require two instructions.

The MicroBlaze processor ABI offers two global small data areas that can each contain up to 64 KB of data. Any memory location within these areas can be accessed using the small data area anchors and a 16-bit immediate value, needing only one instruction for a load or store to the small data area. This optimization can be turned on with the `-mxl-gp-opt` command line parameter. Variables of size less than a certain threshold value are stored in these areas and can be addressed with fewer instructions. The addresses are calculated during the linking stage.



IMPORTANT! *If this option is being used, it must be provided to both the compile and the link commands of the build process for your program. Using the switch inconsistently can lead to compile, link, or runtime errors.*

- **-mno-clearbss:** This option is useful for compiling programs used in simulation.

According to the C language standard, uninitialized global variables are allocated in the `.bss` section and are guaranteed to have the value 0 when the program starts execution. Typically, this is achieved by the C startup files running a loop to fill the `.bss` section with zero when the program starts execution. Optimizing compilers also allocates global variables that are assigned zero in C code to the `.bss` section.

In a simulation environment, the above two language features can be unwanted overhead. Some simulators automatically zero the entire memory. Even in a normal environment, you can write C code that does not rely on global variables being zero initially. This switch is useful for these scenarios. It causes the C startup files to not initialize the `.bss` section with zeroes. It also internally forces the compiler to not allocate zero-initialized global variables in the `.bss` and instead move them to the `.data` section. This option might improve startup times for your application. Use this option with care and ensure either that you do not use code that relies on global variables being initialized to zero, or that your simulation platform performs the zeroing of memory.

- **-mxl-stack-check:** With this option, you can check whether the stack overflows when the program runs.

The compiler inserts code in the prologue of every function, comparing the stack pointer value with the available memory. If the stack pointer exceeds the available free memory, the program jumps to a subroutine `_stack_overflow_exit`. This subroutine sets the value of the variable `_stack_overflow_error` to 1.

You can override the standard stack overflow handler by providing the function `_stack_overflow_exit` in the source code, which acts as the stack overflow handler.

Application Execution Modes

- `-x1-mode-executable`: This is the default mode used for compiling programs with mb-gcc. This option need not be provided on the command line for mb-gcc. This uses the startup file `crt0.o`.
- `x1-mode-bootstrap`: This option is used for applications that are loaded using a bootloader. Typically, the bootloader resides in non-volatile memory mapped to the processor reset vector. If a normal executable is loaded by this bootloader, the application reset vector overwrites the reset vector of the bootloader. In such a scenario, on a processor reset, the bootloader does not execute first (it is typically required to do so) to reload this application and do other initialization as necessary.

To prevent this, you must compile the bootloaded application with this compiler flag. On a processor reset, control then reaches the bootloader instead of the application.

Using this switch on an application that is deployed in a scenario different from the one described above will not work. This mode uses `crt2.o` as a startup file.

- `-x1-mode-novectors`: This option is used for applications that do not require any of the MicroBlaze vectors. This is typically used in standalone applications that do not use any of the processor's reset, interrupt, or exception features. Using this switch leads to smaller code size due to the elimination of the instructions for the vectors. This mode uses `crt3.o` as a startup file.



IMPORTANT! Do not use more than one mode of execution on the command line. You will receive link errors due to multiple definition of symbols if you do so.

Position Independent Code

The GNU compiler for MicroBlaze supports the `-fPIC` and `-fpic` switches. These switches enable position independent code (PIC) generation in the compiler. This feature is used by the Linux operating system only for MicroBlaze to implement shared libraries and relocatable executables. The scheme uses a global offset table (GOT) to relocate all data accesses in the generated code and a procedure linkage table (PLT) for making function calls into shared libraries. This is the standard convention in GNU-based platforms for generating relocatable code and for dynamically linking against shared libraries.

MicroBlaze Application Binary Interface

The GNU compiler for MicroBlaze uses the Application Binary Interface (ABI) defined in the *MicroBlaze Processor Reference Guide* ([UG081](#)). Refer to the ABI documentation for register and stack usage conventions and a description of the standard memory model used by the compiler.

MicroBlaze Assembler

The `mb-as` assembler for the MicroBlaze soft processor supports the same set of options supported by the standard GNU compiler tools. It also supports the same set of assembler directives supported by the standard GNU assembler.

The `mb-as` assembler supports all the opcodes in the MicroBlaze machine instruction set, with the exception of the `imm` instruction. The `mb-as` assembler generates `imm` instructions when large immediate values are used. The assembly language programmer is never required to write code with `imm` instructions. For more information on the MicroBlaze instruction set, refer to the *MicroBlaze Processor Reference Guide* ([UG081](#)).

The `mb-as` assembler requires all MicroBlaze instructions with an immediate operand to be specified as a constant or a label. If the instruction requires a PC-relative operand, then the `mb-as` assembler computes it and includes an `imm` instruction if necessary.

For example, the branch immediate if equal (`beqi`) instruction requires a PC-relative operand.

The assembly programmer should use this instruction as follows:

```
beqi r3, mytargetlabel
```

where `mytargetlabel` is the label of the target instruction. The `mb-as` assembler computes the immediate value of the instruction as `mytargetlabel - PC`.

If this immediate value is greater than 16 bits, the `mb-as` assembler automatically inserts an `imm` instruction. If the value of `mytargetlabel` is not known at the time of compilation, the `mb-as` assembler always inserts an `imm` instruction. Use the `relax` option of the linker to remove any unnecessary `imm` instructions.

Similarly, if an instruction needs a large constant as an operand, the assembly language programmer should use the operand as is, without using an `imm` instruction. For example, the following code adds the constant 200,000 to the contents of register `r3`, and stores the results in register `r4`:

```
addi r4, r3, 200000
```

The `mb-as` assembler recognizes that this operand needs an `imm` instruction, and inserts one automatically.

In addition to the standard MicroBlaze instruction set, the `mb-as` assembler also supports some pseudo-op codes to ease the task of assembly programming. The following table lists the supported pseudo-opcodes.

Table 66: Pseudo-Opcodes Supported by the GNU Assembler

Pseudo Opcodes	Explanation
<code>nop</code>	No operation. Replaced by instruction: <code>or R0, R0, R0</code>
<code>la Rd, Ra, Imm</code>	Replaced by instruction: <code>addik Rd, Ra, imm; = Rd = Ra + Imm;</code>
<code>not Rd, Ra</code>	Replace by instruction: <code>xori Rd, Ra, -1</code>
<code>neg Rd, Ra</code>	Replace by instruction: <code>rsub Rd, Ra, R0</code>
<code>sub Rd, Ra, Rb</code>	Replace by instruction: <code>rsub Rd, Rb, Ra</code>

MicroBlaze Linker Options

The `mb-ld` linker for the MicroBlaze soft processor provides additional options to those supported by the GNU compiler tools. The options are summarized in this section.

- `-defsym _TEXT_START_ADDR=value`: By default, the text section of the output code starts with the base address 0x28. This can be overridden by using the `-defsym _TEXT_START_ADDR` option. If this is supplied to `mb-gcc` compiler, the text section of the output code starts from the given value.

You do not have to use `-defsym _TEXT_START_ADDR` if you want to use the default start address set by the compiler.

This is a linker option and should be used when you invoke the linker separately. If the linker is being invoked as a part of the `mb-gcc` flow, you must use the following option:

```
-Wl, -defsym _TEXT_START_ADDR=value
```

- `-relax`: This is a linker option that removes all unwanted `imm` instructions generated by the assembler. The assembler generates an `imm` instruction for every instruction where the value of the immediate cannot be calculated during the assembler phase. Most of these instructions do not need an `imm` instruction. These are removed by the linker when the `-relax` command line option is provided.

This option is required only when linker is invoked on its own. When linker is invoked through the `mb-gcc` compiler, this option is automatically provided to the linker.

- **-N:** This option sets the text and data section as readable and writable. It also does not page-align the data segment. This option is required only for MicroBlaze programs. The top-level GCC compiler automatically includes this option, while invoking the linker, but if you intend to invoke the linker without using GCC, use this option.

The MicroBlaze linker uses linker scripts to assign sections to memory. These are listed in the following section.

MicroBlaze Linker Script Sections

The following table lists the input sections that are assigned by MicroBlaze linker scripts.

Table 67: Section Names and Descriptions

Section	Description
.vectors.reset	Reset vector code.
.vectors.sw_exception	Software exception vector code.
.vectors.interrupt	Hardware Interrupt vector code.
.vectors.hw_exception	Hardware exception vector code.
.text	Program instructions from code in functions and global assembly statements.
.rodata	Read-only variables.
.sdata2	Small read-only static and global variables with initial values.
.data	Static and global variables with initial values. Initialized to zero by the boot code.
.sdata	Small static and global variables with initial values.
.sbss2	Small read-only static and global variables without initial values. Initialized to zero by boot code.
.sbss	Small static and global variable without initial values. Initialized to zero by the boot code.
.bss	Static and global variables without initial values. Initialized to zero by the boot code.
.heap	Section of memory defined for the heap.
.stack	Section of memory defined for the stack.

Tips for Writing or Customizing Linker Scripts

Keep the following points in mind when writing or customizing your own linker script:

- Ensure that the different vector sections are assigned to the appropriate memories as defined by the MicroBlaze hardware.

- Allocate space in the `.bss` section for stack and heap. Set the `_stack` variable to the location after `_STACK_SIZE` locations of this area, and the `_heap_start` variable to the next location after the `_STACK_SIZE` location. Because the stack and heap need not be initialized for hardware as well as simulation, define the `_bss_end` variable after the `.bss` and `COMMON` definitions.

Note: The `.bss` section boundary does not include either stack or heap.
- Ensure that the variables `_SDATA_START__`, `_SDATA_END__`, `SDATA2_START`, `_SDATA2_END__`, `_SBSS2_START__`, `_SBSS2_END__`, `_bss_start`, `_bss_end`, `_sbss_start`, and `_sbss_end` are defined to the beginning and end of the sections `sdata`, `sdata2`, `sbss2`, `bss`, and `sbss` respectively.
- ANSI C requires that all uninitialized memory be initialized to startup (not required for stack and heap). The standard CRT that is provided assumes a single `.bss` section that is initialized to zero. If there are multiple `.bss` sections, this CRT does not work. You should write your own CRT that initializes all the `.bss` sections.

Startup Files

The compiler includes pre-compiled startup and end files in the final link command when forming an executable. Startup files set up the language and the platform environment before your application code executes. Start up files typically do the following:

- Set up any reset, interrupt, and exception vectors as required.
- Set up stack pointer, small-data anchors, and other registers. Refer to the following table for details.
- Clear the BSS memory regions to zero.
- Invoke language initialization functions, such as C++ constructors.
- Initialize the hardware subsystem. For example, if the program is to be profiled, initialize the profiling timers.
- Set up arguments for the main procedure and invoke it.

Similarly, end files are used to include code that must execute after your program ends. The following actions are typically performed by end files:

- Invoke language cleanup functions, such as C++ destructors.
- Deinitialize the hardware subsystem. For example, if the program is being profiled, clean up the profiling sub-system.

The following table lists the register names, values, and descriptions in the C runtime files.

Table 68: Register Initialization in C Runtime Files

Register	Value	Description
r1	_stack-16	The stack pointer register is initialized to point to the bottom of the stack area with an initial negative offset of 16 bytes. The 16 bytes can be used for passing in arguments.
r2	_SDA2_BASE	_SDA2_BASE_ is the read-only small data anchor address.
r13	_SDA_BASE_	_SDA_BASE is the read-write small data anchor address.
Other registers	Undefined	Other registers do not have defined values.

The following subsections describe the initialization files used for various application modes. This information is for advanced users who want to change or understand the startup code of their application.

For MicroBlaze, there are two distinct stages of C runtime initialization. The first stage is primarily responsible for setting up vectors, after which it invokes the second stage initialization. It also provides exit stubs based on the different application modes.

First Stage Initialization Files

- `crt0.o`: This initialization file is used for programs which are to be executed in standalone mode, without the use of any bootloader or debugging stub. This CRT populates the reset, interrupt, exception, and hardware exception vectors and invokes the second stage startup routine `_crtinit`. On returning from `_crtinit`, it ends the program by infinitely looping in the `_exit` label.
- `crt1.o`: This initialization file is used when the application is debugged in a software-intrusive manner. It populates all the vectors except the breakpoint and reset vectors and transfers control to the second-stage `_crtinit` startup routine.
- `crt2.o`: This initialization file is used when the executable is loaded using a bootloader. It populates all the vectors except the reset vector and transfers control to the second-stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the reset vector is not populated, on a processor reset, control is transferred to the bootloader, which can reload and restart the program.
- `crt3.o`: This initialization file is employed when the executable does not use any vectors and wishes to reduce code size. It populates only the reset vector and transfers control to the second stage `_crtinit` startup routine. On returning from `_crtinit`, it ends the program by infinitely looping at the `_exit` label. Because the other vectors are not populated, the GNU linking mechanism does not pull in any of the interrupt and exception handling related routines, thus saving code space.

Second Stage Initialization Files

According to the C standard specification, all global and static variables must be initialized to 0. This is a common functionality required by all the CRTs above. Another routine, `_crtinit`, is invoked. The `_crtinit` routine initializes memory in the `.bss` section of the program. The `_crtinit` routine is also the wrapper that invokes the main procedure. Before invoking the main procedure, it might invoke other initialization functions. The `_crtinit` routine is supplied by the startup files described below.

- `crtinit.o`:

This default, second stage, C startup file performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes “constructor” functions (`_init`).
4. Sets up the arguments for main and invokes main.
5. Invokes “destructor” functions (`_fini`).
6. Invokes `_program_clean` and returns.

- `pgcrtinit.o`:

This second stage startup file is used during profiling, and performs the following steps:

1. Clears the `.bss` section to zero.
2. Invokes `_program_init`.
3. Invokes `_profile_init` to initialize the profiling library.
4. Invokes “constructor” functions (`_init`).
5. Sets up the arguments for main and invokes main.
6. Invokes “destructor” functions (`_fini`).
7. Invokes `_profile_clean` to cleanup the profiling library.
8. Invokes `_program_clean`, and then returns.

- `sim-crtinit.o`:

This second-stage startup file is used when the `-mno-clearbss` switch is used in the compiler, and performs the following steps:

1. Invokes `_program_init`.
2. Invokes “constructor” functions (`_init`).
3. Sets up the arguments for main and invokes main.

4. Invokes “destructor” functions (`_fini`).
 5. Invokes `_program_clean`, and then returns.
- `sim-pgcrtinit.o`:

This second stage startup file is used during profiling with the `-mno-clearbss` switch, and performs the following steps in order:

1. Invokes `_program_init`.
2. Invokes `_profile_init` to initialize the profiling library.
3. Invokes “constructor” functions (`_init`).
4. Sets up the arguments for and invokes `main`.
5. Invokes “destructor” functions (`_fini`).
6. Invokes `_profile_clean` to cleanup the profiling library.
7. Invokes `_program_clean`, and then returns.

Other Files

The compiler also uses certain standard start and end files for C++ language support.

These are `crti.o`, `crtbegin.o`, `crtend.o`, and `crtn.o`. These files are standard compiler files that provide the content for the `.init`, `.fini`, `.ctors`, and `.dtors` sections.

Modifying Startup Files

The initialization files are distributed in both precompiled and source form with Vivado. The pre-compiled object files are found in the compiler library directory. Sources for the initialization files for the MicroBlaze GNU compiler can be found in the `<XILINX_>/Vitis/<version>/data/embeddedsw/lib/microblaze/src/` directory, where `<XILINX_>` is the Vivado installation path and `<version>` is the release version of the Vitis software platform.

To fulfill a custom startup file requirement, you can take the files from the source area and include them as a part of your application sources. Alternatively, you can assemble the files into `.o` files and place them in a common area. To refer to the newly created object files instead of the standard files, use the `-B directory -name` command line option while invoking `mb-gcc`.

To prevent the default startup files from being used, use the `-nostartfiles` on the final compile line.

Note: The miscellaneous compiler standard CRT files, such as `crti.o`, and `crtbegin.o`, are not provided with source code. They are available in the installation to be used as is. You might need to bring them in on your final link command.

Reducing the Startup Code Size for C Programs

If your application has stringent requirements on code size for C programs, you might want to eliminate all sources of overhead. This section describes how to reduce the overhead of invoking the C++ constructor or destructor code in a C program that does not require that code. You might be able to save approximately 220 bytes of code space by making the following modifications:

1. Follow the instructions for creating a custom copy of the startup files from the installation area, as described in the preceding sections. Specifically, copy over the particular versions of `crti.s` and `xcrtinit.s` that suit your application. For example, if your application is being bootstrapped and profiled, copy `crt2.s` and `pg-crtinit.s` from the installation area.
2. Modify `pg-crtinit.s` to remove the following lines:

```
brlid r15, __init
/* Invoke language initialization functions */
nop
```

and

```
brlid r15, __fini
/* Invoke language cleanup functions */
nop
```

This avoids referencing the extra code usually pulled in for constructor and destructor handling, reducing code size.

3. Compile these files into `.o` files and place them in a directory of your choice, or include them as a part of your application sources.
4. Add the `-nostartfiles` switch to the compiler. Add the `-B` directory switch if you have chosen to assemble the files in a particular folder.
5. Compile your application.

If your application is executing in a different mode, then you must pick the appropriate CRT files based on the description in [Startup Files](#).

Compiler Libraries

The `mb-gcc` compiler requires the GNU C standard library and the GNU math library.

Precompiled versions of these libraries are shipped with Vivado. The CPU driver for MicroBlaze copies over the correct version, based on the hardware configuration of MicroBlaze. To manually select the library version that you would like to use, look in the following folder:

```
$XILINX/_gnu/microblaze/<platform>/microblaze-xilinx-elf/lib
```

The filenames are encoded based on the compiler flags and configurations used to compile the library. For example, `libc_m_bs.a` is the C library compiled with hardware multiplier and barrel shifter enabled in the compiler.

The following table shows the current encodings used and the configuration of the library specified by the encodings.

Table 69: Encoded Library Filenames on Compiler Flags

Encoding	Description
<code>_bs</code>	Configured for barrel shifter.
<code>_m</code>	Configured for hardware multiplier.
<code>_p</code>	Configured for pattern comparator.

Of special interest are the math library files (`libm*.a`). The C standard requires the common math library functions (`sin()` and `cos()`, for example) to use double-precision floating point arithmetic. However, double-precision floating point arithmetic might not be able to make full use of the optional, single-precision floating point capabilities in available for MicroBlaze.

The newlib math libraries have alternate versions that implement these math functions using single-precision arithmetic. These single-precision libraries might be able to make direct use of the MicroBlaze processor hardware floating point unit (FPU) and could therefore perform better.

If you are sure that your application does not require standard precision, and you want to implement enhanced performance, you can manually change the version of the linked-in library.

By default, the CPU driver copies the double-precision version (`libm*_fpd.a`) of the library into your IP integrator project.

To get the single precision version, you can create a custom CPU driver that copies the corresponding `libm*_fps.a` library instead. Copy the corresponding `libm*_fps.a` file into your processor library folder (such as `microblaze_0/lib`) as `libm.a`.

When you have copied the library that you want to use, rebuild your application software project.

Thread Safety

The MicroBlaze processor C and math libraries distributed with Vivado are not built to be used in a multi-threaded environment. Common C library functions such as `printf()`, `scanf()`, `malloc()`, and `free()` are not thread-safe and causes unrecoverable errors in the system at runtime. Use appropriate mutual exclusion mechanisms when using the Vivado libraries in a multi-threaded environment.

Command Line Arguments

The MicroBlaze processor programs cannot take command line arguments. The command line arguments `argc` and `argv` are initialized to 0 by the C runtime routines.

Interrupt Handlers

Interrupt handlers must be compiled in a different manner than normal sub-routine calls. In addition to saving non-volatiles, interrupt handlers must save the volatile registers that are being used. Interrupt handlers should also store the value of the machine status register (RMSR) when an interrupt occurs.

- `interrupt_handler`: To distinguish an interrupt handler from a sub-routine, `mb-gcc` looks for an attribute (`interrupt_handler`) in the declaration of the code. This attribute is defined as follows:

```
void function_name () __attribute__ ((interrupt_handler));
```

Note: The attribute for the interrupt handler is to be given *only* in the prototype and *not* in the definition.

Interrupt handlers might also call other functions, which might use volatile registers. To maintain the correct values in the volatile registers, the interrupt handler saves all the volatiles, if the handler is a non-leaf function.

Note: Functions that have calls to other sub-routines are called non-leaf functions.

Interrupt handlers are defined in the Microprocessor Software Specification (MSS) files. These definitions automatically add the attributes to the interrupt handler functions. The interrupt handler uses the instruction `rtid` for returning to the interrupted function.

- **save_volatile**: The MicroBlaze compiler provides the attribute `save_volatile`, which is similar to the `interrupt_handler` attribute, but returns using `rtsd` instead of `rtid`. This attribute saves all the volatiles for non-leaf functions and only the used volatiles in the case of leaf functions.

```
void function_name () __attribute__((save_volatile));
```

- **fast_interrupt**: The MicroBlaze compiler provides the attribute `fast_interrupt`, which is similar to the `interrupt_handler` attribute. On fast interrupt, MicroBlaze jumps to the interrupt routine address instead jumping to the fixed address 0x10.

Unlike a normal interrupt, when the attribute `fast_interrupt` is used on a C function, MicroBlaze saves only minimal registers.

```
void function_name () __attribute__((fast_interrupt));
```

Table 70: Use of Attributes

Attributes	Functions
interrupt_handler	This attribute saves the machine status register and all the volatiles, in addition to the non-volatile registers. <code>rtid</code> returns from the interrupt handler. If the interrupt handler function is a leaf function, only those volatiles which are used by the function are saved.
save_volatile	This attribute is similar to <code>interrupt_handler</code> , but it uses <code>rtsd</code> to return to the interrupted function, instead of <code>rtid</code> .
fast_interrupt	This attribute is similar to <code>interrupt_handler</code> , but it jumps directly to the interrupt routine address instead of jumping to the fixed address 0x10.

Arm Compiler Usage and Options

1. Arm® Cortex®-A9 targets can be compiled using the arm-none-eabi toolchain
2. Arm® Cortex-A53 targets can be compiled using the aarch64-none-elf toolchain
3. Arm® Cortex-R5F targets can be compiled using the armr5-none-eabi toolchain

Arm® Cortex® A9 targets can be compiled using the `arm-none-eabi` toolchain. The `arm-none-eabi` toolchain contains the complete GNU toolchain including all of the following components:

- Common startup code sequence
- GNU binary utilities (binutils)
- GNU C compiler (GCC)
- GNU C++ compiler (G++)
- GNU C++ runtime library (libstdc++)
- GNU debugger (GDB)
- Newlib C library

Usage

Note: Cortex®-R5F and A53 toolchains can be used in a similar way.

Compiling

```
arm-none-eabi-gcc -c file1.c -I<include_path> -o file1.o
arm-none-eabi-gcc -c file2.c -I<include_path> -o file2.o
```

Linking

```
arm-none-eabi-gcc -Wl,-T -Wl,lscript.ld -L<libxil.a path> -o
"App.elf" file1.o file2.o -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
```

For descriptions of flags used in the commands above, refer to the compiler help, using any of the following commands:

- `--help`
- `-v --help`
- `--target-help`

Compiler Options

Other GNU compiler options that can be applied using Arm®-related flags can be found on the [GNU website](#). These flags can be used in the steps above, as required. All the Arm® GCC compiler options are listed at the link above. However, actual support depends on the target in use (Arm Cortex-A9, Cortex-A53, Cortex®-A72, and Cortex-R5F in this case) and on the compiler toolchain.

Other Notes

C++ Code Size

The GCC toolchain combined with the latest open source C++ standard library (`libstdc++-v3`) might be found to generate large code and data fragments as compared to an equivalent C program. A significant portion of this overhead comes from code and data for exception handling and runtime type information. Some C++ applications do not require these features.

To remove the overhead and optimize for size, use the `-fno-exceptions` and/or the `-fno-rtti` switches. This is recommended only for advanced users who know the requirements of their application and understand these language features. Refer to the [GCC documentation](#) for more specific information on available compiler options and their impact.

C++ programs might have more intensive dynamic memory requirements (stack and heap size) due to more complex language features and library routines. Many of the C++ library routines can request memory to be allocated from the heap. Review your heap and stack size requirements for C++ programs to ensure that they are satisfied.

C++ Standard Library

The C++ standard defines the C++ standard library. A few of these platform features are unavailable on the default AMD Vivado™ software platform. For example, file I/O is supported in only a few well-defined `STDIN/STDOUT` streams. Similarly, locale functions, thread-safety, and other such features might not be supported.

Note: The C++ standard library is not built for a multi-threaded environment. Common C++ features such as `new` and `delete` are not thread safe. Use caution when using the C++ standard library in an operating system environment.

For more information on the GNU C++ standard library, refer to the documentation available on the GNU website.

Position Independent Code (Relocatable Code)

The MicroBlaze™ processor compilers support the `-fPIC` switch to generate position independent code. While both these features are supported in the AMD compiler, they are not supported by the rest of the libraries and tools, because Vivado only provides a standalone platform. No loader or debugger can interpret relocatable code and perform the correct relocations at runtime. These independent code features are not supported by the AMD libraries, startup files, or other tools. Third-party OS vendors could use these features as a standard in their distribution and tools.

Other Switches and Features

Other switches and features might not be supported by the AMD Vivado compilers and/or platform, such as `-fprofile-arcs`. Some features might also be experimental in nature (as defined by open source GCC) and could produce incorrect code if used inappropriately. Refer to the [GCC documentation](#) for more information on specific features.

Embedded Design Tutorials

The following hardware specific embedded design tutorials are available for embedded software designers.

- *Zynq 7000 SoC: Embedded Design Tutorial* ([UG1165](#))
- *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* ([UG1209](#))
- *Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* ([UG1305](#))

The software features are demonstrated in the Vitis tutorial under embedded design section. See *Vitis Unified Software Platform Tutorials Landing Page* ([UG1605](#)).

Section VIII

Drivers and Libraries

Drivers and libraries are hosted on the AMD wiki. You can access them with the following links:

- [Bare-metal Drivers and Libraries](#)
- [Linux Drivers](#)

Additional Resources and Legal Notices

Finding Additional Documentation

Technical Information Portal

The AMD Technical Information Portal is an online tool that provides robust search and navigation for documentation using your web browser. To access the Technical Information Portal, go to <https://docs.amd.com>.

Documentation Navigator

Documentation Navigator (DocNav) is an installed tool that provides access to AMD Adaptive Computing documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the AMD Vivado™ IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, click the **Start** button and select **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Note: For more information on DocNav, refer to the *Documentation Navigator User Guide* ([UG968](#)).

Design Hubs

AMD Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- Go to the [Design Hubs](#) web page.

Support Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Support](#).

Revision History

Getting Started with Vitis Revision History

The following table shows the revision history for [Section I: Getting Started with Vitis](#).

Section	Revision Summary
05/30/2024 Version 2024.1	
General updates	General updates.
12/13/2023 Version 2023.2	
Migrating from the Classic Vitis IDE to Vitis Unified IDE	Added supporting links.
10/18/2023 Version 2023.2	
N/A	Updated for Vitis Unified IDE.
07/26/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
06/12/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
05/16/2023 Version 2023.1	
Vitis Software Platform Release Notes	Fixed issues and updated for 2023.1.
01/02/2023 Version 2022.2	
General updates	Editorial updates only. No technical content updates.
12/23/2022 Version 2022.2	
N/A	No changes to this section.
10/19/2022 Version 2022.2	
Vitis Software Platform Release Notes	Updated for 2022.2.
04/26/2022 Version 2022.1	
N/A	No changes to this section.
12/15/2021 Version 2021.2	
N/A	No changes to this section.
10/22/2021 Version 2021.2	
Vitis Software Platform Release Notes	Updated release notes.
Installation	Updated installation information.

Using the Vitis IDE Revision History

The following table shows the revision history for [Section II: Using the Vitis Unified IDE](#).

Section	Revision Summary
05/30/2024 Version 2024.1	
Vitis Unified IDE View and Feature	Added two new features: file change indicator and New feature page.
Target Platform	Added platform creation advanced debug options.
Running and Debugging Application Components under a System Project Together	Added hierarchical debug support.
Multi-Cable and Multi-Device Support	Added Multi-Cable and Multi-Device Support.
12/13/2023 Version 2023.2	
Keyboard Shortcuts, Command Palette, and Quick Find	Added description of keyboard shortcut of Quick Find.
10/18/2023 Version 2023.2	
Section II: Using the Vitis Unified IDE	Added information to migrate from Classic Vitis IDE to Vitis Unified IDE.
07/26/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
06/12/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
05/16/2023 Version 2023.1	
N/A	No changes to this section.
01/02/2023 Version 2022.2	
General updates.	Editorial updates only. No technical content updates.
12/23/2022 Version 2022.2	
N/A	No changes to this section.
10/19/2022 Version 2022.2	
Develop	Updated screenshots.
04/26/2022 Version 2022.1	
Debug Perspective in Unified IDE	New topic.

Bootgen Revision History

The following table shows the revision history for [Section III: Bootgen Tool](#).

Section	Revision Summary
05/30/2024 Version 2024.1	
BIF Attribute Reference	Added information for <ul style="list-style-type: none"> • enable_auth_opt • optionaldata
12/13/2023 Version 2023.2	
N/A	Editorial changes only.

Section	Revision Summary
10/18/2023 Version 2023.2	
Boot Image Layout	Updated GUI images and general bug fixes.
07/26/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
06/15/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
05/16/2023 Version 2023.1	
imagedstore	Added topic.
01/02/2023 Version 2022.2	
General updates	Editorial updates only. No technical content updates.
12/23/2022 Version 2022.2	
SSIT Support	General updates.
12/14/2022 Version 2022.2	
SSIT Support	Updated for Versal HBM.
10/19/2022 Version 2022.2	
SSIT Support	Added SSI technology Authentication flow.
N/A	Bug fixes.
04/26/2022 Version 2022.1	
Using Bootgen GUI	Updated Images
SSIT Support	Updated for Versal Premium
BIF Attribute Reference	Updated BIF Attributes
Command Reference	Updated Command References

Vitis Python CLI

The following table shows the revision history for [Section IV: Vitis Python CLI](#).

Section	Revision Summary
05/30/2024 Version 2024.1	
Python API: A command-line tool for creating and managing projects in Vitis	Added new section.
12/13/2023 Version 2023.2	
N/A	Added new chapter added.

Software Command-Line Tool Revision History

The following table shows the revision history for [Section V: Software Command-Line Tool](#).

Section	Revision Summary
05/30/2024 Version 2024.1	
N/A	No updates.

Section	Revision Summary
12/13/2023 Version 2023.2	
N/A	Editorial changes only.
10/18/2023 Version 2023.2	
N/A	Updated for 2023.2 with editorial changes.
07/26/2023 Version 2023.1	
N/A	No changes to this section.
06/12/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
05/16/2023 Version 2023.1	
Memory and Register accesses from XSCT	Added topic.
Loading U-Boot over JTAG	Added topic for U-Boot flow using XSCT.
	Added support for including multiple dtsi files, instead of only one, in the device-tree.
01/02/2023 Version 2022.2	
General updates	Editorial updates only. No technical content updates.
12/23/2022 Version 2022.2	
N/A	No changes to this section.
10/19/2022 Version 2022.2	
Section V: Software Command-Line Tool	<ul style="list-style-type: none"> Fixed regression in platform config and platform list commands. Supported template XSAs with createdts command. Supported user dtsi files with createdts command. Supported QEMU args files for Versal DC platforms. Added new option to stapl config command to generate crc. Cleared the entire TCM memory in Versal if any of the elf sections use tcm. Added AI Engine to processors list returned by getprocessors command if the xsa contains AI Engine.
04/26/2022 Version 2022.1	
N/A	No changes to this section.

GNU Compiler Tools Revision History

The following table shows the revision history for [Section VI: GNU Compiler Tools](#).

Section	Revision Summary
05/30/2024 Version 2024.1	
N/A	No changes to this section.
12/13/2023 Version 2023.2	
Section VII: Embedded Design Tutorials	Added supporting link.
10/18/2023 Version 2023.2	
N/A	No changes to this section.

Section	Revision Summary
07/26/2023 Version 2023.1	
N/A	No changes to this section.
06/12/2023 Version 2023.1	
General updates	Editorial updates only. No technical content updates.
05/16/2023 Version 2023.1	
Embedded GNU Toolchain Details	All tool chain updated. GCC to 12.2 Version and GDB 12.1 Version.
01/02/2023 Version 2022.2	
General updates	Editorial updates only. No technical content updates.
12/23/2022 Version 2022.2	
N/A	No changes to this section.
10/19/2022 Version 2022.2	
N/A	No changes to this section.
04/26/2022 Version 2022.1	
General updates	Minor editorial changes.

Please Read: Important Legal Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. **THIS INFORMATION IS PROVIDED "AS IS."** AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2019-2024 Advanced Micro Devices, Inc. AMD, the AMD Arrow logo, UltraScale, UltraScale+, Versal, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the US and/or elsewhere. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.